MATCH: A MATLAB Compiler For Configurable Computing Systems

P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann
M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare
A. Nayak, S. Periyacheri, M. Walkden
Electrical and Computer Engineering
Northwestern University
2145 Sheridan Road
Evanston, IL-60208
banerjee@ece.nwu.edu
(847) 491-3641

Abstract

[1] Configurable computing systems constitute a new class of computing and communication systems which are composed of configurable hardware capable of system-level adaptation. The objective of the MATCH (MATlab Compiler for Heterogeneous computing systems) compiler project at Northwestern University is to make it easier for the users to develop efficient codes for configurable computing systems. Towards this end we are implementing and evaluating an experimental prototype of a software system that will take MATLAB descriptions of various embedded systems applications, and automatically map them on to a configurable computing environment consisting of field-programmable gate arrays, embedded processors and digital signal processors built from commercial off-the-shelf components. In this paper, we provide an overview of the MATCH compiler and discuss the testbed which is being used to demonstrate our ideas of the MATCH compiler. We present preliminary experimental results on some benchmark MATLAB programs with the use of the MATCH compiler.

KEYWORDS: Configurable computing, FPGA, DSP, embedded, MATLAB, Compiler, Libraries, Heterogeneous processors, Type Inferencing, Testbed, Benchmarks, Performance Evaluation.

[1]This research was supported in part by the Defense Advanced Research Projects Agency under contract F30602-98-2-0144 and in part by the National Science Foundation under grant NSF CDA-9703228.

Paper Submitted to IEEE Computer Magazine, Aug. 1999.

1 Introduction

Configurable computing systems constitute a new class of computing and communication systems which are composed of configurable hardware capable of system-level adaptation [6, 5]. One can visualize such systems to consist of field-programmable gate arrays (FPGA) and field-programmable interconnect chips. However, purely FPGA-based systems are usually unsuitable for complete algorithm implementation. In most computations there is a large amount of code that is executed relatively rarely, and attempting to map all of these functions into reprogrammable logic would be very logic-inefficient. Also, reconfigurable logic is much slower than a processor's built-in functional units for standard computations such as floating point and complex integer arithmetic, variable length shifts, and others.

The solution to this dilemma is to combine the advantages of both microprocessor based embedded systems (distributed and heterogeneous), specialized processors such as digital signal processors (DSPs), and FPGA resources into a single system. The microprocessors are used to support the bulk of the functionality required to implement an algorithm, while the reconfigurable logic is used to accelerate only the most critical computation kernels of the program. This would make a typical configurable computing system a heterogeneous array of embedded processors, DSPs and FPGAs.

A key question that needs to be addressed is how to map a given computation on such a heterogeneous architecture without expecting the application programmer to get into the low level details of the architecture or forcing him/her to understand the finer issues of mapping the applications on such a distributed heterogeneous platform. Recently, high-level languages such as MATLAB have become popular in prototyping algorithms in domains such as signal and image processing, the same domains which are the primary users of embedded systems. MATLAB provides a very high level language abstraction to express computations in a functional style which is not only intuitive but also concise. However, currently no tools exist that can take such high-level specifications and generate low level code for such a heterogeneous testbed automatically.

We would like to motivate our work with an example hyperspectral image processing application from the NASA Goddard Space Center. The hyperspectral image classification application attempts to classify a hyperspectral image in order to make it more useful for analysis by humans. Example uses are to determine the type of terrain being represented in the image: land, swamp, ocean. In 1999, NASA will launch the first Earth Observation Satellite, the Terra. The Terra satellite is projected to average 918 GBytes of data per day. If one were to download the raw data to earth, it would fill up NASA's entire data holdings (125,000 GBytes so far) in 6 months. Hence it is important to perform computation on the data on board the satellite itself before sending down the data to earth. The need for configurable computing in this environment is that there is a lot of instrument dependent processing that needs to be performed. The computations on the data involve several algorithms, and the algorithms often change over the lifetime of the instrument. Figure 1 shows an overview of the NASA's hyperspectral application. Currently, NASA scientists have to write all the code for execution on embedded processors in assembly language or C, and the code for execution on

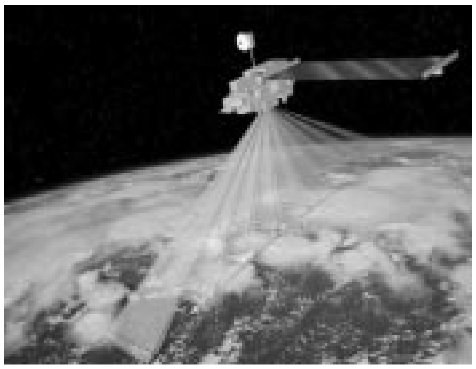


Figure 1: NASA's Hyperspectral application. The need for configurable computing in this environment is that there is a lot of instrument dependent processing that needs to be performed. The computations on the data involve several algorithms, and the algorithms often change over the lifetime of the instrument.

the FPGAs in structural or RTL VHDL code. The application scientists that are developing algorithms for such a satellite are experts in signal and image processing and often prototype their algorithms in languages such as MATLAB. Hence there is a need to develop tools to convert high-level descriptions of algorithms in MATLAB and compile them automatically to run on a heterogeneous network of embedded processors, DSP processors, and FPGAs.

The objective of the MATCH (MATlab Compiler for Heterogeneous computing systems) compiler project at Northwestern University is to make it easier for the users to develop efficient codes for configurable computing systems. Towards this end we are implementing and evaluating an experimental prototype of a software system that will take MATLAB descriptions of various embedded systems applications, and automatically map them on to a configurable computing environment consisting of field-programmable gate arrays, embedded processors and digital signal processors built from commercial off-the-shelf components. An overview of the easy-to-use programming environment that we are trying to accomplish through our MATCH compiler is shown in Figure 2. The goal of our compiler is to generate efficient code automatically for such a heterogeneous target that will be within a factor of 2-4 of the best manual approach with regard to two optimization objectives: (1) Optimizing resources (such as type and number of processors, FPGAs, etc) under performance constraints (such as delays, and throughput) (2) Optimizing performance under resource constraints.

The paper is organized as follows. Section 2 provides an overview of the testbed which is being used to demonstrate our ideas of the MATCH compiler. We describe the various components of the MATCH compiler in Section 3. We present preliminary experimental results of our compiler in Section 4. We compare our

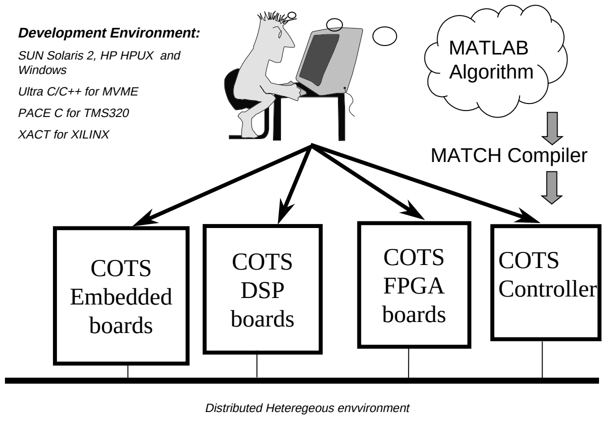


Figure 2: A graphical representation of the objectives of our MATCH compiler. The objective is to have an user develop various algorithms in image and signal processing in MATLAB, specify the constraints regarding delays and throughputs, and for the MATCH compiler to automatically partition and schedule the computations on different portions of the testbed

work with other related research in Section 5, and conclude the paper in Section 6.

2 Overview of MATCH Testbed

The testbed that we have designed to work with the MATCH project consists of four types of compute resources. These resources are realized by using off-the-shelf boards plugged into a VME cage. The VME bus provides the communication backbone for some control applications. In addition to the reconfigurable resources, our testbed also incorporates conventional embedded and DSP processors to handle special needs of some of the applications. Real life applications often have parts of the computations which may not be ideally suited for the FPGAs. They could be either control intensive parts or could be even complex floating point applications. Such computations are performed by these embedded and DSP processors. An overview of the testbed is shown in Figure 3

We use an off-the-shelf multi-FPGA board from Annapolis Microsystems as the reconfigurable part of our testbed. This $WildChild^{TM}$ board (refer to Figure 4) has 8 Xilinx 4010 FPGAs (each with 400 CLBs, 512KB local memory) and a Xilinx 4028 FPGAs (with 1024 CLBs, 1MB local memory). All these FPGAs can communicate among themselves either via a on board 36 bit wide systolic bus or via a crossbar. An on-board micro controller interfaces to the Force V host (explained later) via the VME bus. With the help of this controller, the host can effectively configure the FPGAs. Further, the local memories of the FPGAs can be accessed by the Host controller, aiding in block data transfers to and from the board.

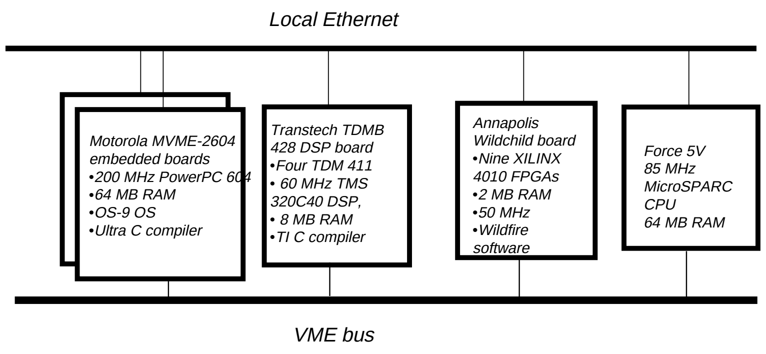


Figure 3: Overview of the Testbed to Demonstrate the MATCH Compiler

A Transtech TDM-428 board is used as a DSP resource. This board has four Texas Instruments TMS320C40 processors (each running at 60MHz, with 8MB RAM) interconnected by an on board 8 bit wide 20MB/sec communication network. One of these processors can communicate with the host processor via the VME bus interface. The other general purpose compute resource employed in the MATCH testbed is a set of Motorola MVME2604 boards. Each of these boards hosts a PowerPC-604 processor (each running at 200 MHz, with 64 MB local memory) running Microware's OS-9 operating system. These processors can communicate among themselves via a 100BaseT ethernet interface.

A Force 5V board with MicroSPARC-II processor running Solaris Operating system forms one of the compute resources that also plays the role of a main controller of the testbed. This board can communicate with other boards either via the VME bus or via the ethernet interface. This processor moreover functions as a master to the FPGA and DSP boards that act as attached processors.

3 The MATCH Compiler

We will now discuss various aspects of the MATCH compiler that automatically translates the MATLAB programs and maps them on to different parts of the target system shown in Figure 3. The overview of the compiler is shown in Figure 5.

MATLAB is basically a *function oriented* language and most of the MATLAB programs can be written

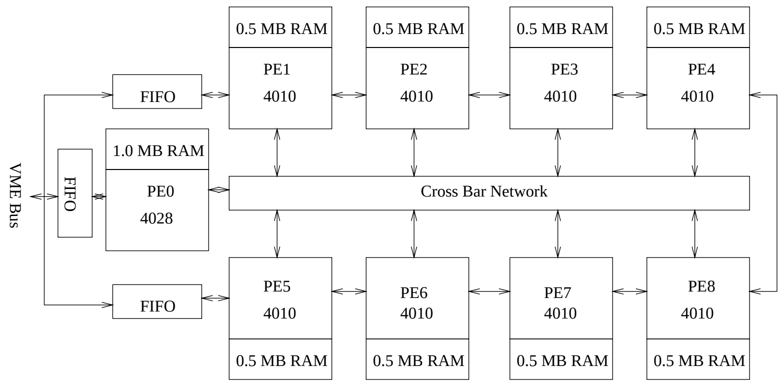


Figure 4: Reconfigurable part of the MATCH testbed

using predefined functions. These functions can be primitive functions or application specific. In a sequential MATLAB program, these functions are normally implemented as sequential codes running on a conventional processor. In the MATCH compiler however, these functions need to be implemented on different types of resources (both conventional and otherwise) and also some of these need to be parallel implementations to take best advantage of the parallelism supported by the underlying target machine.

MATLAB also supports language constructs using which a programmer can write conventional procedural style programs (or parts of it) using loops, vector notation and the like. Our MATCH compiler needs to automatically translate all such parts of the program into appropriate sequential or parallel codes. MATLAB being a *dynamically typed* language, poses several problems to a compiler. One of them being the well known *type inferencing problem*. The compiler has to figure out not only whether a variable was meant to be a floating point variable, but also the number of dimensions and extent in each dimension if the variable happens to be an array. Our compiler provides mechanisms to automatically perform such inferencing which is a crucial component of any compiler for a dynamically typed language.

The maximum performance from a parallel heterogeneous target machine such as the one shown in Figure 3 can only be extracted by efficient mapping of various parts of the MATLAB program onto the appropriate parts of the target. The MATCH compiler incorporates automatic mechanisms to perform such mapping. It also provides ways using which an experienced programmer well versed with the target characteristics can guide the compiler to fine tune the mapping in the form of directives.

3.1 The Compilation Overview

The first step in producing parallel code from a MATLAB program involves parsing the input MATLAB program based on a formal grammar and building an abstract syntax tree. Figure 6 shows a graphical view of the hierarchy captured by the grammar. An example MATLAB code and the corresponding abstract syntax tree is also shown. After the abstract syntax tree is constructed the compiler invokes a series of *phases*. Each phase processes the abstract syntax tree by either modifying it or annotating it with more

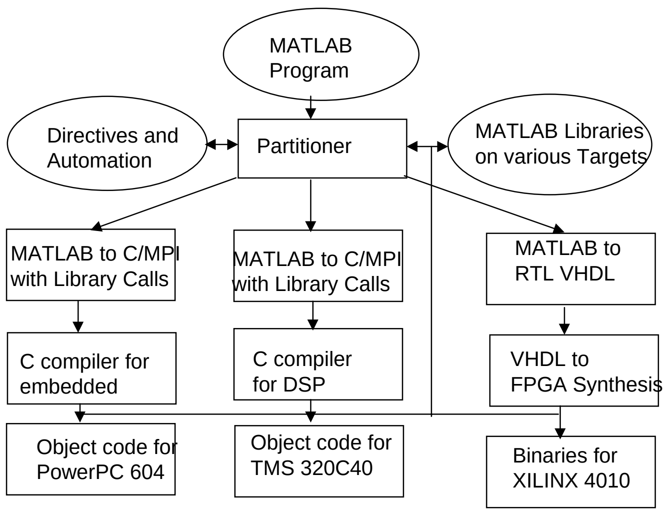

MATLAB Program
Directives and Automation
Partitioner
MATLAB Libraries on various Targets
MATLAB to C/MPI with Library Calls
MATLAB to C/MPI with Library Calls
MATLAB to RTL VHDL
C compiler for embedded
C compiler for DSP
VHDL to FPGA Synthesis
Object code for PowerPC 604
Object code for TMS 320C40
Binaries for XILINX 4010

Figure 5: The MATCH Compiler Components

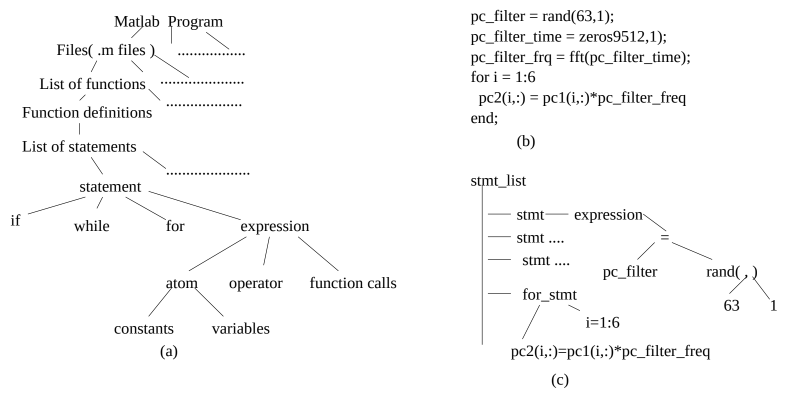
information.

Using rigorous data/control flow analysis and taking cues from the programmer directives (explained in Section 3.6.2), this AST is partitioned into one or more sub trees. The nodes corresponding to the predefined library functions directly map on to the respective targets and any procedural style code is encapsulated as a user defined procedure. The main thread of control is automatically generated for the Force V processor which keeps making remote procedure calls to these functions running on the processor (or processors) onto which they are mapped. Figure 7 shows various transformations done to the MATLAB AST to finally convert it into a parallel program running on the target.

In the following sections we go into the details of these aspects of the MATCH compiler.

3.2 MATLAB Functions on FPGAs

In this section we describe our effort in the development of various MATLAB libraries on the Wildchild FPGA board described earlier and shown in Figure 4. These functions are developed in Register Transfer Level (RTL) VHDL using the Synplify logic synthesis tool from Synplicity to generate gate level netlists, and the Alliance place-and-route tools from Xilinx. Some of the functions we have developed on the FPGA board include matrix addition, matrix multiplication, one dimensional FFT and FIR and IIR Filters. In each case we have developed C program interfaces to our MATCH compiler so that these functions can be



```
pc_filter = rand(63,1);
pc_filter_time = zeros9512,1);
pc_filter_frq = fft(pc_filter_time);
for i = 1:6
  pc2(i,:) = pc1(i,:)*pc_filter_freq
end;
```

Figure 6: Abstract Syntax Tree: (a) The hierarchy captured by the formal grammar (b) A sample MATLAB code (c) Abridged syntax tree for the MATLAB code.

called from the host controller. In the sidebar subsection we discuss the implementation of one of these functions namely the IIR filter function as a typical example.

3.2.1 IIR and FIR Filter (A sidebar section in the final version)

Filtering is one of the most common operations performed in signal processing. Most filters belong to one of two classes - FIR for Finite Impulse Response and IIR for Infinite Impulse Response filter. While the actual implementation of these two classes of filters is different, mathematically they are quite similar, and the FIR filter can be done using the more general IIR filter. We have implemented MATLAB library functions for FIR or IIR filtering of up to order 64 on vectors of maximum size 250,000 elements. The data precision used is 8 bits fixed-point in fractional 2's complement format.

There are several ways to implement a IIR filter, many of which build higher order filters out of smaller ones. The cascaded form of the IIR filter lends well to implementation on the multi-FPGA architecture of the WILDCHILD system due to the presence of near-neighbor communication capability via the systolic bus. Several FPGAs can be strung together in series to implement the required filter operation. The primary constraint in our implementation has been the resource requirements of the filter unit on each FPGA. Since the FPGAs on the WILDCHILD board are very modest, we were able to fit only a 1-tap filter (first order filter or 'integrator') on each FPGA.

Each 1-tap filter unit on the FPGAs is reversible. It can accept input from the left or right systolic bus and output the result on the right or left bus respectively. The input vector is initially stored in the local memory of PE1 and fed into the 1-tap filter on PE1. The partially filtered data then moves on to PE2, then to PE3 and so on in a pipelined manner until it reaches PE8 which writes its result to its local memory completing an 8-tap filter. For higher-order filters, all PEs are reversed and the partially filtered data is read from PE8's local memory, passed through PE7, PE6 and so on until it reaches PE1 where it gets written to

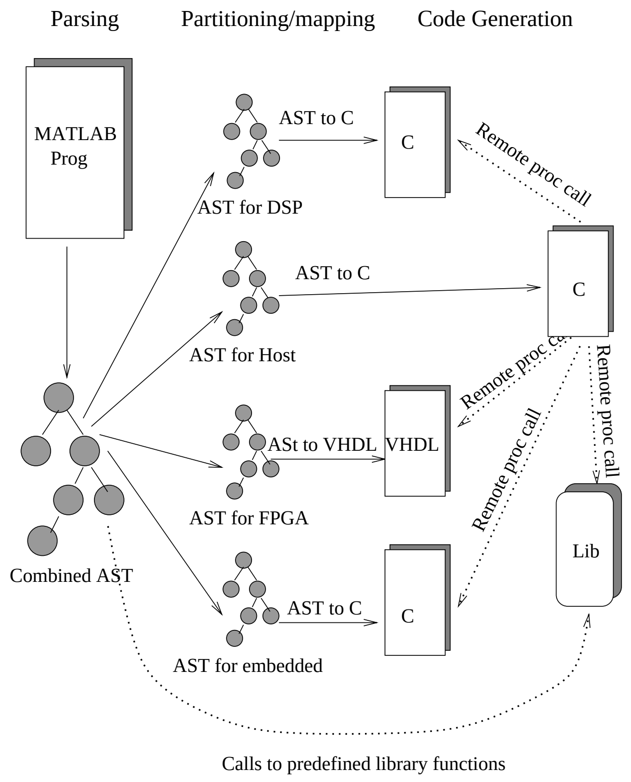


Figure 7: Program Transformations

PE1's local memory completing a 16-tap filter. This process of passing data back and forth can be repeated as many times as required to implement filters of any size. PE0 is not used in this design and is configured as a 'blank' FPGA.

The performance characteristics of this filter implementation for various number of taps and various data sizes is shown in Table 1. These characterizations are used by the automated mapping algorithm of the MATCH compiler described in Section 3.6.1.

3.3 MATLAB Functions on DSPs

In this section we will describe our effort in the development of various MATLAB library functions on the DSPs. These functions are developed on the Transtech DSP boards utilizing multiple DSPs using message-passing among multiple processors in C using our own custom implementation of MPI. We subsequently used the PACE C compiler from Texas Instruments to generate the object code for the TMS320C40 processors. Our current set of functions includes real and complex matrix addition, real and complex matrix multiplication ,one and two dimensional FFT. Each of these libraries has been developed with a variety of data distributions such as blocked, cyclic and block-cyclic distributions. In the sidebar section, we go through the implementation details of one of these functions (FFT) as a typical example.

Table 1: IIR/FIR filter library function characterizations on the Wildchild FPGA board of the MATCH testbed. Runtimes in milli-seconds are shown for various number of taps and various data sizes.

Filter Taps	Vector Size	Config Time	Download + Readback	Compute
16	16K	2600	132+15=147	3
16	64K	2600	188+58=246	13
16	256K	2600	440+230=670	52
64	16K	2600	132+15=147	13
64	64K	2600	188+58=246	52
64	256K	2600	440+230=670	210
256	16K	2600	132+15=147	52
256	64K	2600	188+58=246	210
256	256K	2600	440+230=670	840

Table 2: 2-D FFT library function characterizations on a 2-D image on the DSP board of the MATCH testbed. Runtimes in seconds are shown for various image sizes and various processor sizes/configurations and data distributions.

	Processor configs		
Image Size	1x1	2x1	4x1
16 x 16	0.007	0.006	0.005
64 x 64	0.151	0.089	0.057
128 x 128	0.668	0.378	0.233
256 x 256	3.262	1.799	1.06

3.3.1 2-dimensional Fast Fourier Transform (A sidebar section in the paper)

The 2-dimensional Fast Fourier Transform (FFT) takes a 2-D image of size N * N, and performs 1-D FFT on each row and 1-D FFT on each column of the array. To ensure best performance, the distribution of the input data among the cluster of DSPs is very important. Though a row-wise data distribution is best suited for FFT computation along the rows with the entire row being present within the same processor, this generates a lot of communication when one performs the column-wise FFTs unless one performs a transpose of the intermediate data. Our implementation of FFT is generic enough to handle FFT when data is distributed differently.

The performance characteristics of this 2-D FFT implementation on the DSP board for various number of processors and various data distributions, and various data sizes is shown in Table 2. The table shows the results for a 1D FFT with along the rows with the data distributed in a cyclic(4),cyclic(4) manner. The speedup is around 3 on 4 processors. These characterizations are used by the automated mapping algorithm of the MATCH compiler described in Section 3.6.1.

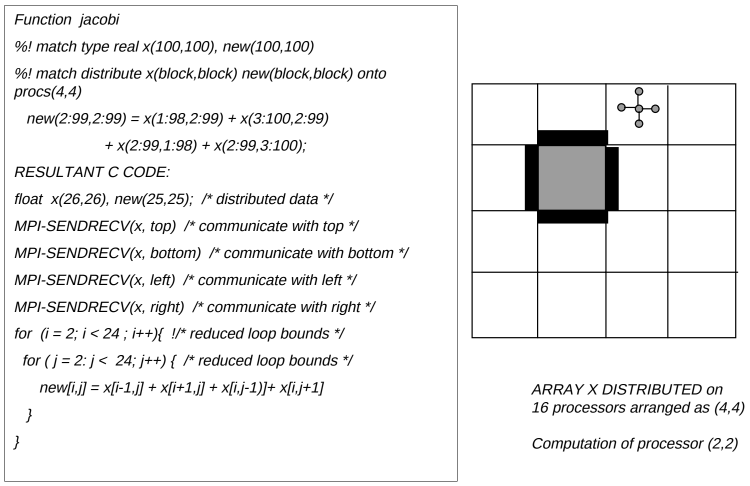
3.4 Automatic Generation of User functions

Since MATLAB allows procedural style of programming using constructs such as loops and control flow, the parts of the program written in such a style may not map to any of the predefined library functions. All such fragments of the program need to be translated appropriately depending on the target resource onto which they are mapped. As shown in Figure 5, we wish to generate C code for the DSP and embedded processors and VHDL code for the FPGAs. In most cases we need to translate them into parallel versions to take advantage of multiple resources that can exploit data parallelism.

The particular paradigm of parallel execution that we have presently considered is the *single-program-multiple-data* (SPMD) model. In this particular scenario, the compiler must decide on which processors to perform the actual compute operations and various candidate schemes exist to guide the compiler in making its decision. One possible scheme is called the *owner-computes rule* in which operations on a particular data element are executed by only those processors that actually "own" the data element. This is the way computations are currently assigned to processors in our compiler. Ownership is determined by the alignments and distributions that the data is subjected to. Though automated procedures do exist which discover suitable alignments and distributions that the data must be subjected to so as to minimize inter-processor communication, our current infrastructure relies on user-provided data alignment and data distribution directives for this information.

Under the owner-computes rule, assignment statements are executed by the processor that owns the data element being assigned to. We illustrate the compilation process through the example shown in Figure 8 for the Jacobi iterative method for solving the Laplace equation. The Jacobi method updates all elements of a *new* array using the average value of all its top, bottom, left and right neighbors of the old array x. The following points are to be noted. First, corresponding to the array assignment statement, its scalarized C equivalent has a set of nested `for` loops whose loop bounds have to be determined by the data distribution among the processors. Also the computation of elements at the border may need data from neighboring processors, which are obtained as part of four vectorized messages called MPI-SENDRECV(). The compilation process described here is analogous to that followed by High-Performance Fortran compilers when they generate message passing Fortran code.

The SPMD code generation process must partition the loop bounds in the loop nest so that each processor executes only those iterations for which `new[i,j]` resides on it. This is done by taking into account the alignment and distribution directives. The computation partitioning is achieved using a linear algebra framework which is sufficiently sophisticated to handle affine subscript and loop bound expressions that may also involve compile-time unknowns. This framework also helps in finding out the data needed from other processors and also the exact way of sharing them.



```
Function  jacobi
%! match type real x(100,100), new(100,100)
%! match distribute x(block,block) new(block,block) onto
procs(4,4)
   new(2:99,2:99) = x(1:98,2:99) + x(3:100,2:99)
                  + x(2:99,1:98) + x(2:99,3:100);
RESULTANT C CODE:
float  x(26,26), new(25,25);  /* distributed data */
MPI-SENDRECV(x, top)  /* communicate with top */
MPI-SENDRECV(x, bottom)  /* communicate with bottom */
MPI-SENDRECV(x, left)  /* communicate with left */
MPI-SENDRECV(x, right)  /* communicate with right */
for  (i = 2; i < 24 ; i++){  !/* reduced loop bounds */
  for ( j = 2: j <  24; j++) {  /* reduced loop bounds */
      new[i,j] = x[i-1,j] + x[i+1,j] + x[i,j-1)]+ x[i,j+1]
   }
}
```

ARRAY X DISTRIBUTED on
16 processors arranged as (4,4)
Computation of processor (2,2)

Figure 8: Example automatic generation of SPMD code for the Jacobi computation. Since the two arrays new and x of size 100 X 100 are distributed among a 4 X 4 array of processors, each processor works on a sub-array of size 25 X 25. Any non-local data that is accessed has to be obtained through message passing prior to execution of the computation.

3.5 Type Inferencing

In this section, we discuss the mechanisms used by the MATCH compiler to perform type inferencing. When possible, our compiler tries to infer type and shape of variables using automatic inferencing techniques. Often, it may not be possible to infer these attributes, in which case our compiler takes the help of user *directives* which allow the programmer to explicitly *declare* the type/shape information.

3.5.1 Automatic Inferencing

First, we convert the expressions in the original MATLAB source to a sequence of subexpressions, all having the single operator form. We then infer the shapes of the various subexpressions using a shape algebra. The determination of a variable's shape is often non-trivial because of its potentially symbolic nature.
If e is an arbitrary MATLAB expression, we define its shape $\sigma(e)$ as follows:

$$\sigma(e) \leftarrow \begin{cases} \sigma_D(e) \text{ where } \sigma_D(e) \in \mathcal{Z} \text{ and } \sigma_D(e) \geq 2, \\ \sigma_P(e), \\ \widehat{\sigma_T}(e) \leftarrow \begin{pmatrix} p_1 & 0 & 0 & \cdots & 0 \\ 0 & p_2 & 0 & \cdots & 0 \\ 0 & 0 & p_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & p_{\sigma_D(e)} \end{pmatrix} \text{ where } p_i \in \mathcal{Z} \text{ and } p_i \geq 0 \ \forall \ 1 \leq i \leq \sigma_D(e). \end{cases} \tag{1}$$

Thus, the shape of a MATLAB expression consists of its dimensionality $\sigma_D(e)$, a boolean expression $\sigma_P(e)$ referred to as its shape-predicate, and finally, its shape-tuple matrix $\widehat{\sigma_T}(e)$. The shape-tuple matrix is a square diagonal matrix having the extents of the MATLAB expression e as its principal diagonal elements. These elements can be symbolic and their validity is ensured by the shape-predicate which must be true at run-time.

It is possible to formulate a shape algebra which expresses each of the three components of a MATLAB expression's shape in terms of the shape components of the expression's subexpressions. For instance, given the MATLAB assignment $\mathtt{c} \leftarrow \mathtt{a} + \mathtt{b}$, we have the following:

$$\begin{array}{l} \sigma_D(c) \leftarrow \max(\sigma_D(a), \sigma_D(b)), \\ \sigma_P(c) \leftarrow (\alpha(a) = 1) \vee (\alpha(b) = 1) \vee (\widehat{\sigma_T}^*(a) = \widehat{\sigma_T}^*(b)), \\ \widehat{\sigma_T}^*(a)\alpha(b) + \widehat{\sigma_T}^*(b)(1 - \alpha(b)). \end{array}$$

In the above, $\alpha(a)$ and $\alpha(b)$ are scalars that are unity if and only if the MATLAB expressions a and b are themselves scalars. The symbolic expressions for $\alpha(a)$ and $\alpha(b)$ can also be formulated in terms of the shape components of a and b. The matrices $\widehat{\sigma_T}^*(a)$ and $\widehat{\sigma_T}^*(b)$ are basically $\widehat{\sigma_T}(a)$ and $\widehat{\sigma_T}(b)$ "expanded" to the dimensionality of the result (i.e., $\sigma_D(c)$).

3.5.2 User Guided Inferencing

User guided type inferencing in the MATCH compiler is done by taking cues from the programmer by means of *directives*. The directives for match compiler start with %!match and hence appear as comments to other MATLAB interpreters/compilers. An extensive set of directives have been designed for the MATCH compiler. Some examples of these directives are:

%!match TYPE integer a
$a = 2$ % this specifies 'a' as being a scalar
%!match TYPE real a(512,512) , b(100)
$[a, b] = foo(1, 2)$
% this specifies a as 512x512 real matrix, b is a 100 element row vector.
%!match TYPE real a(512,512) , b(N)
$[a, b] = foo(1, 2)$
% Here N must be a constant known at compile-time.
%!match TYPE real a(unknown), b(unknown,10)

$[a, b] = foo(1, 2)$

3.6 Mapping onto the Target

3.6.1 Automated Mapping

When possible, the MATCH compiler tries to automatically map the user program on to the target machine taking into account the specified timing constraints, device capabilities and costs. The automatic mapping is formulated as a mixed integer linear programming problem with two optimization objectives: (1) Optimizing resources (such as type and number of processors, FPGAs, etc) under performance constraints (such as delays, and throughput) (2) Optimizing performance under resource constraints. The performance characterization of the predefined library functions and the user defined procedures guide this automatic mapping. Examples of performance characterizations are illustrated in Table 1 for the FPGA board and Table 2 for the DSP board in our MATCH testbed. Figure 9 shows some of the steps involved in automatic mapping. Figure 9(a) shows the control-data flow graph of a given MATLAB program. Each node of the graph consists of MATLAB functions which can have several implementations on various resources such as single or multiple FPGAs and DSP processors. These functions are characterized in terms of the costs on various resources and execution times in Figure 9(b). Finally, using our mixed integer linear programming formulation and solution for the time constrained resource optimization problem results in the resource selection, pipelining, and scheduling, shown in Figure 9(c).

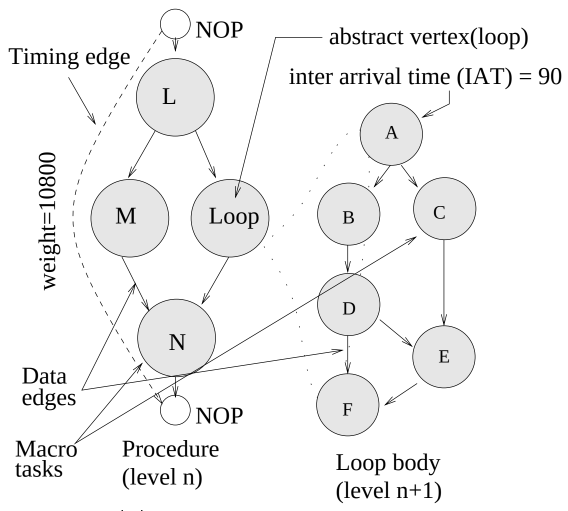
3.6.2 User Guided Mapping

In cases where such an automatic mapping is not satisfactory or if the programmer is in a better position to guide the compiler, special user *directives* are provided for this purpose. These directives describe the target architectures to the compiler, the availability of predefined libraries on them and other relevant characteristics. Some examples of such directives supported by the MATCH compiler are:

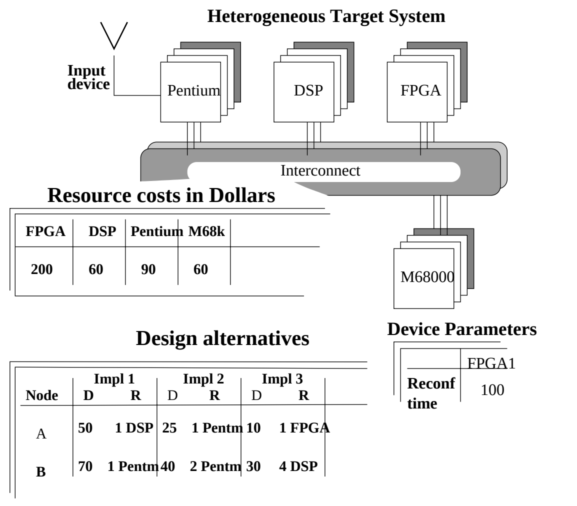
```
%!match USE FUNCTION fpga-fft
fft(1:n)
%!match BIND TASK filter TO PROCESSOR[1..10]
filter(1:m)
%!match USE PROCESSOR TYPE "DSP"
c = a * b
```

3.7 Complete Code Generation

After generating the ASTs for each of the individual parts of the original MATLAB program, these ASTs are suitably translated for appropriate target processors. Depending on the mapping (performed as discussed in Section 3.6), the targets for each of these ASTs could be different. The ASTs corresponding to FPGAs are



(a) Internal representation.



Resource costs in Dollars

FPGA	DSP	Pentium	M68k
200	60	90	60

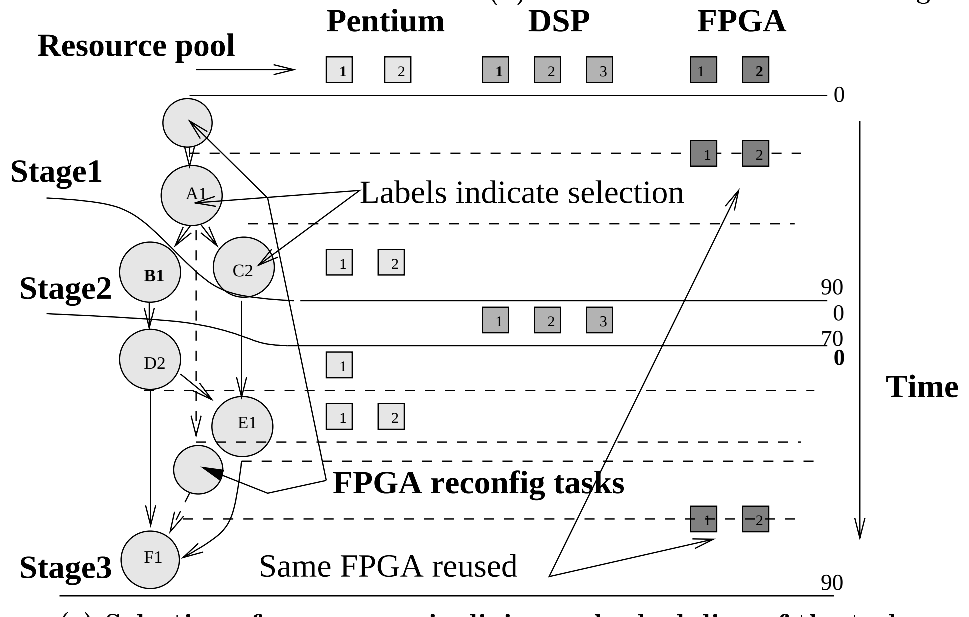
Design alternatives

	Impl 1		Impl 2		Impl 3	
Node	D	R	D	R	D	R
A	50	1 DSP	25	1 Pentm	10	1 FPGA
B	70	1 Pentm	40	2 Pentm	30	4 DSP

Device Parameters

	FPGA1
Reconf time	100

(b) Available resources and design alternatives



(c) Selection of resources, pipelining and scheduling of the tasks

Figure 9: Automatic Mapping

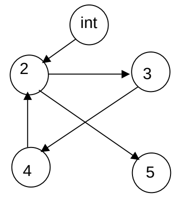
translated to RTL VHDL and those corresponding to Host/DSP/Embedded processors are translated into equivalent C language programs. Finally, these generated programs are compiled using the respective target compilers to generate the executable/configuration bit streams.

3.7.1 Code Generation for FPGAs

Each user function is converted into a process in VHDL. Each scalar variable in MATLAB is converted into a variable in VHDL. Each array variable in MATLAB is assumed to be stored in a RAM adjacent to the FPGA, hence a corresponding read or write function of a memory process is called from the FPGA computation process. Control statements such as IF-THEN-ELSE constructs in MATLAB are converted into corresponding IF-THEN-ELSE constructs in VHDL. Assignment statements in MATLAB are converted into variable assignment statements in VHDL. Loop control statements are converted into a finite state machine as shown in Figure 10.

MATLAB PROGRAM

```
function sum()
sum = 0;
for i =1:100
  sum = sum + a(i);
end
```



State diagram of finite state machine

Register Transfer Level VHDL PROGRAM

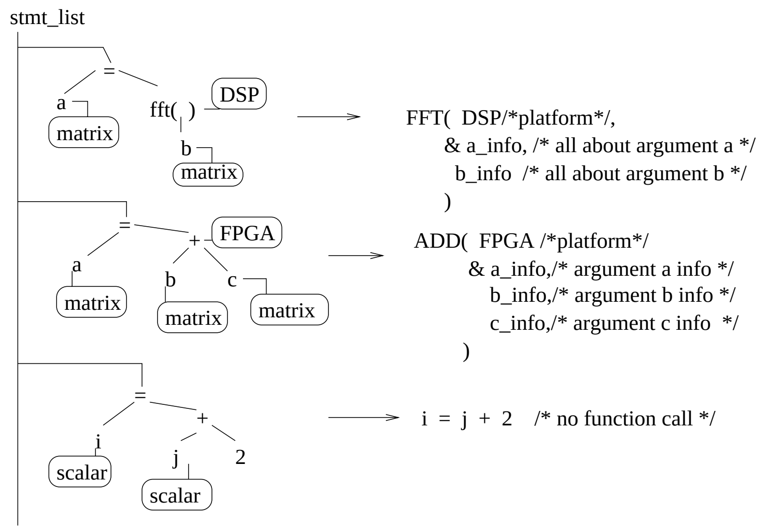
```
process sum
if rising_edge(clock)
case state is
when int: sum := 0; i := 0;
             state := 2;
when 2: if (I < 100) state := 3;
           else state := 5;
when 3:  RAMaddr := a + I;
           Readmemory := 1;
           state := 4;
when 4:
           sum := sum + RAM_out;
           i := i + 1;
           state := 2;
when 5:
           result_sum := sum;
           state := init;
end process
```

Figure 10: Example compilation of a MATLAB program with loops into RTL VHDL. On the top left we show an example MATLAB code which performs a summation of elements of an array. We show the state diagram of a finite state machine to perform the same computation in the bottom left. On the right we show the corresponding RTL VHDL code.

For each loop statement, we create a finite state machine with four states. The first state performs the initialization of loop control variables and any variables used inside the loop. The second state checks if the loop exit condition is satisfied. If condition is valid, it transfers control to state 4, which is the end of the loop. If condition is not valid, it transfers control to state 3, which performs the execution of statements in the loop body. If there is an array access statement (either read or write), one needs to generate extra states to perform the memory read/write from external memory and wait the correct number of cycles.

3.7.2 Code Generation for Host/DSP/Embedded

Once the abstract syntax tree is constructed, generating the code is quite straightforward. A reverse post-order traversal of the abstract syntax tree is done. At nodes corresponding to operators and function calls, the annotated information about the operands and the operator/function are checked. Depending upon the annotated information a call to a suitable C function is inserted that accomplishes the task of the operator/function call in the MATLAB program. For example, to invoke the *fft* function on the cluster of DSP processors, the compiler generates a call to a *wrapper function fft(DSP,....)*, instead of the set of actual calls needed to invoke the fft on the DSP processors cluster. This wrapper contains the mechanism to invoke the function on all the available platforms. Similarly, to invoke the *fft* on the FPGAs instead of the DSPs, the compiler just has to generate the call *fft(FPGA,...)*.



```
FFT(  DSP/*platform*/,
      & a_info, /* all about argument a */
        b_info  /* all about argument b */
      )

ADD(  FPGA /*platform*/
        & a_info,/* argument a info */
           b_info,/* argument b info */
           c_info,/* argument c info  */
        )

i  =  j  +  2    /* no function call */
```

Figure 11: Code Generation: The *_info represent structures that all necessary information such as type,shape,precision, actual data itself etc about the variable. The fields of the structure get defined along with the definition of the variable. These informations are used where ever the variable is used henceforth. For operators, function calls are generated only when operands are not scalar.

4 Experimental Results

We have implemented a preliminary version of the MATCH compiler. In this section we will report results of the MATCH compiler on some benchmark MATLAB programs.

4.1 Matrix Multiplication

We first report on results of the simple matrix multiplication function on various parts of the testbed. We perform the same matrix multiplication function on three targets on the testbed. The following MATLAB code shown below represents the matrix multiplication test benchmark. We use directives to map the same matrix multiplication computation to three targets in our heterogeneous testbed. The compiler calls the appropriate library functions and the related host code.

```
%!match TYPE real a(256,256)
a = rand(256,256)
%!match TYPE real b(256,256)
b = rand(256,256)
%!match TYPE real c(256,256)
%!match USE PROCESSOR TYPE "FORCE"
c = a * b
%!match USE PROCESSOR TYPE "DSP"
c = a * b
%!match PRECISION 8 bits a,b,c
%!match USE PROCESSOR TYPE "FPGA"
c = a * b
end
```

The results of executing the code on the testbed are shown in Table 3. The column shown as "Force" refers to a matrix multiplication library running on the Force board using the RTEExpress library [3] from Integrated Sensors Inc., using 32 bit real numbers. The column shown as "DSP" refers to a matrix multiplication library written in C using one processor on the Transtech DSP board, using 32 bit real numbers. The column shown as "FPGA" refers to a matrix multiplication library function written in VHDL in the Wildchild FPGA board, using 8 bit fractional numbers. It can be seen that even including the FPGA configuration and data download times it is faster to perform matrix multiplication on the Wildchild FPGA board. It should be noted however that the FPGA board is operating at a smaller clock cycle (20 MHz) instead of the Force board running at 85 MHz and the DSP board running at 60 MHz. However the FPGA board has more parallelism since it has 8 FPGAs working in parallel; also the data precision on the FPGA computation is only 8 bits while the Force board and DSP boards are operating on 32 bit integer numbers. The numbers in parenthesis under the column for FPGA refers to the FPGA configuration and data read and write times off the board.

4.2 Fast Fourier Transform

We next report on results of a one-dimensional Fast Fourier Transform function on various parts of the testbed. We perform the same FFT function on four targets on the testbed using a program similar to the previous matrix multiplication example.

Table 3: Comparisons of runtimes in seconds of the matrix multiplication benchmark on various targets of the testbed

	Execution Time(in secs)		
Size	Force RTE	DSP	FPGA
	(85 MHz, 32 bit)	(60 MHz, 32 bit)	(20 Mhz, 8 bit)
64X64	0.36	0.08	0.002 (0.038)
128X128	2.35	0.64	0.015 (0.082)
248X248	16.48	4.6	0.103 (0.246)
496X496	131.18	36.7	0.795 (0.961)

Table 4: Comparisons of runtimes in seconds of the FFT benchmark on various targets of the testbed. The numbers in parenthesis under the column for FPGA refers to the FPGA configuration and data read and write times off the board.

	Execution Time(in secs)		
Size	Force RTE	DSP	FPGA
	(85 MHz, 32 bit)	(60 MHz, 32 bit)	(9 MHz, 8 bit)
128	0.62	0.668	0.00005 (0.51)
256	2.61	3.262	0.00013 (0.51)

The results are shown in Table 4. The column shown as "Force" refers to the FFT running on the Force board with the RTEExpress Library [3] from ISI, using 32 bit real numbers. The column shown as "DSP" refers to the FFT written in C using one processor on the Transtech DSP board, using 32 bit real numbers. The column shown as "FPGA" refers to the FFT written in VHDL in the Wildchild FPGA board, using 8 bit fractional numbers. It can be seen that even including the FPGA configuration and data download times it is faster to perform the FFT on the Wildchild FPGA board. It should be noted however that the FPGA board is operating at a smaller clock cycle (9 MHz) instead of the Force board running at 85 MHz and the DSP board running at 60 MHz.

4.3 Image Correlation

After investigating simple MATLAB functions, we now look at slightly more complex MATLAB programs. One benchmark that we investigated is the image correlation benchmark whose code is shown below. The MATLAB program takes two 2-dimensional image data, performs a 2-dimensional FFT on each, multiplies the result and performs an inverse 2-dimensional FFT on the result, to get the correlation of two images. The MATLAB program annotated with varius directives appears as follows. The type and shape directives specify the size and dimensions of the arrays. The USE directives specify where each of the library functions should be executed. It specifies that the first FFT should be executed on the FPGA board, and the second FFT should be executed on the DSP board, and the matrix multiplication and inverse FFT should be executed on the host Force board using an RTE Express Library [3]from Integrated Sensors Inc.

```
%!match TYPE real x
for x=1:12
%!match TYPE real image1(256,256)
image1 = rand(256,256);
%!match TYPE real image2(256,256)
image2 = rand(256,256);
%!match TYPE REAL m1(256,256)
%!match USE PROCESSOR TYPE "DSP"
m1 = fft(image1);
%!match TYPE REAL m2(256,256)
%!match USE PROCESSOR TYPE "FPGA"
m2 = fft(image2);
%!match TYPE REAL result(256,256)
%!match DISTRIBUTE result(CYCLIC,*) ONTO PROCS(2,2)
!match USE PROCESSOR TYPE "RTE"
result = ifft(m1 * m2)
end
```

The performance of this correlation benchmark using the Mathworks MATLAB 5.2 MCC compiler on a SUN Ultra 5 workstation running Solaris 2.6 and the MATCH testbed is shown in Figure 12.

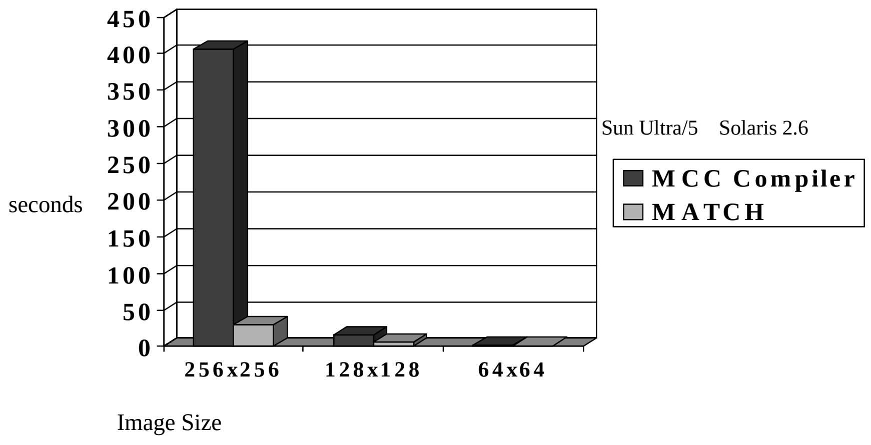


Figure 12: Results the MATHWORKS MCC and MATCH compiler on Correlation Application on a SUN ULtra 5 workstation.

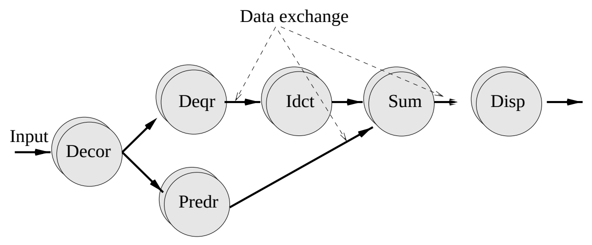


Figure 13: Control and Data Flow Graph of MPEG Decoder

Table 5: MPEG decoder design using heterogeneous resources

Delay (msecs.)	IAT	Cost in $ Greedy	MILP	cost reduction
110	110	375	135	64%
90	90	450	225	50%
100	50	495	315	36%
90	45	515	390	24%
75	25	540	540	0%
60	20	680	680	0%
60	15	805	775	4%
50	10	1145	1145	0%

4.4 MPEG Decoder Design

We have applied our automatic mapping algorithm using mixed integer linear programming (MILP) to the design of MPEG decoder using a heterogeneous set of resources to implement each subtask. The MATLAB program for the MPEG decoder application has six functions called DECOR, DEQUAR, PREDR, IDCT, SUM and DISP. Figure 13 shows the control-dataflow graph of the MPEG program. We assume that there are implementations of each of these functions on single and multiple embedded processors, DSPs and FPGAs of the testbed. Table 5 shows the results of such a mapping for various timing constraints. As can be seen, our MILP algorithm produces results superior to a simple greedy heuristic based mapping.

4.5 Hyperspectral Image Processing

We would like to report the results of one complete application on our testbed described in the introduction. The hyperspectral image classification application from NASA attempts to classify a hyperspectral image in order to make it more useful for analysis by humans. Example uses are to determine the type of terrain being represented: land, swamp, ocean. The algorithm involves the use of a probabilistic neural network computation in order to transform an image into k classes. The MATLAB version of the main part of the code is shown below.

```
for p=1:rows*cols
```

$$f(\vec{X} \mid S_k) = \frac{1}{(2\pi)^{d/2}\sigma^d} \frac{1}{P_k} \sum_{i=1}^{P_k} \exp\left[-\frac{(\vec{X}-\vec{W}_{ki})^T(\vec{X}-\vec{W}_{ki})}{2\sigma^2} \right]$$

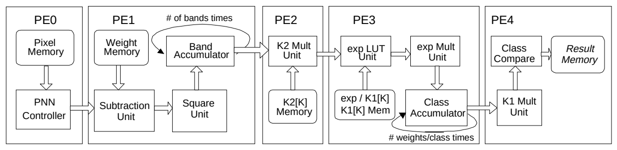


Figure 14: Mapping of Hyperspectral Application on Four FPGAs in MATCH Testbed

```
   % load pixel to process
   pixel = data( (p-1)*bands+1:p*bands );
   class_total = zeros(classes,1);
   class_sum   = zeros(classes,1);
   % class loop
   for c=1:classes
      class_total(c) = 0;
      class_sum(c) = 0;
      % weight loop
      for w=1:bands:pattern_size(c)*bands-bands
         weight = class(c,w:w+bands-1);
         class_sum(c) = exp( -(k2(c)*sum( (pixel-weight').^2 ))) + class_sum(c);
      end
      class_total(c) = class_sum(c) * k1(c);
   end
   results(p) = find( class_total == max( class_total ) )-
1;
end
```

The MATLAB code was mapped onto a set of five FPGAs on the Wildchild Board in our MATCH testbed as shown in Figure 14.

The performance of the application on an HP C-180 workstation with a 180 MHz PA-8000 CPU and 128 MB of memory and the WIldchild FPGA board with five FPGAs is shown in Figure 15. When the application is coded in MATLAB in an iterative manner, it can process 1.6 pixels of the image per second. When the application is coded in MATLAB using vectorization, it can process 35.4 pixels of the image per second. When it is coded in Java, it can process 149 pixels per second, and when coded in C, it can process 364 pixels per second. However, when the application is coded in VHDL and executed on the Annapolis

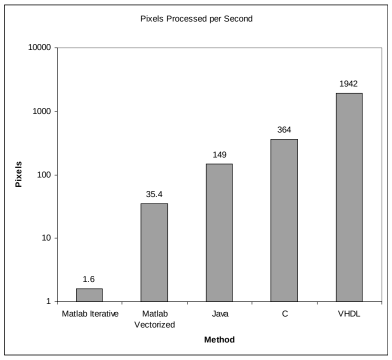


Figure 15: Performance of Hyperspectral Image Processing on Configurable Computing Testbed

Wildchild board with five Xilinx 4010 FPGAs, it can process 1942 pixels per second. Hence one can see the benefits of going from a general purpose processor to a configurable computing system.

We finally show the actual results of an image classification performed on the image in Figure 16 on the configurable computing testbed.

5 Related Work

In this section we review related work in the configurable computing area. These consist of hardware approaches combining the technologies of processors and FPGAs, and software approaches to mapping applications to configurable computing platforms.

5.1 Hardware Developments

The RAW project [10] at MIT is an attempt at building a simple and wire-efficient architecture that scales with increasing VLSI gate densities. A RAW machine is configurable in that it exposes its low-level hardware details completely to the software system, eliminating the traditional compact instruction- set interface (ISA) between the hardware and the compiler.

The BRASS group [9] at the University of California, Berkeley, is investigating the integration of processors and reconfigurable logic. Their first pass at such an architecture was the GARP which combined a MIPS-II processor with a fine-grained FPGA coprocessor on the same die. They are also working on more efficient reconfigurable array designs, techniques to simplify and accelerate the mapping process, and strategies for building reconfigurable applications.

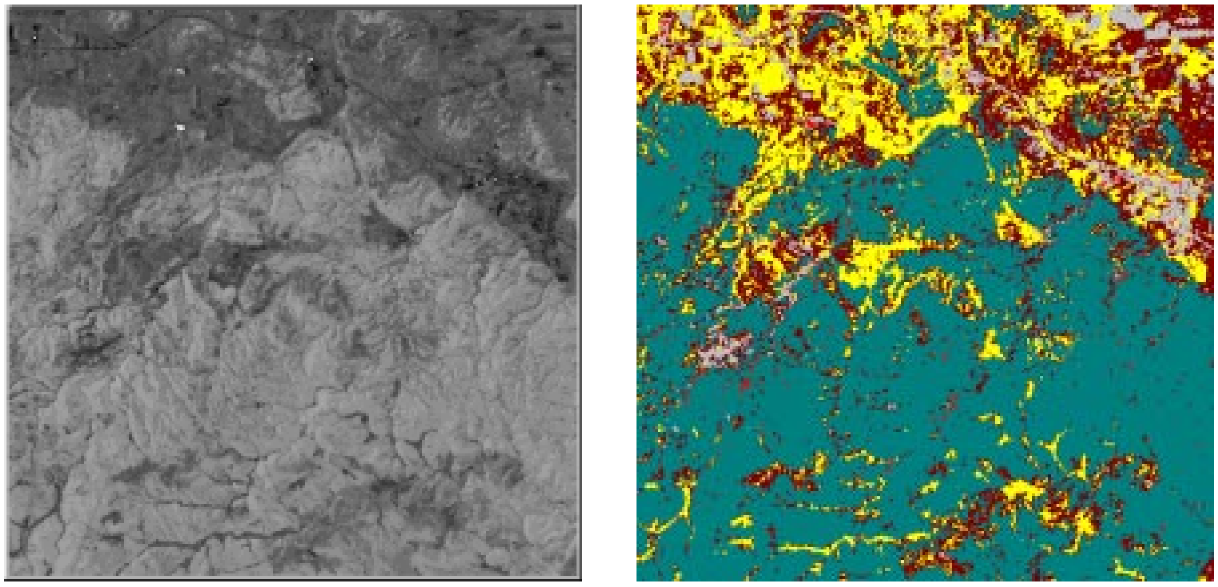

(a) Raw Image (b) Processed Image

Figure 16: Sample output of the raw image and the processed and classified image from the hyper-spectral image classification application.

The Chimaera [8] project is based upon creating a new hardware system consisting of a microprocessor with an internal reconfigurable functional unit. This solution takes advantage of the microprocessor's ability to do the bulk of the work, while the reconfigurable unit performs the compute intensive kernels.

The NAPA1000 Adaptive Processor from National Semiconductor [7] features a merging of FPGA and RISC processor technology. It uses an array of 1024 FPGA based simple processors as a co-processor to the RISC processor to exploit fine grain parallelism in certain applications.

There have been several companies that have developed products which are coprocessor boards that contain FPGAs, embedded processors and memories, and can be attached to the PCI bus or a VME bus. They include the Virtual Computer Corporation's (VCC) the H.O.T. Works PCI Board which contains a Xilinx 6200 FPGA; the ACEcard II from TSI Labs which consists of two Xilinx 4035 FPGAs and a microSPARC II CPU with 64MB of DRAM which attaches to a PCI bus; the WILDFORCE, WILDCHILD and WILDSTAR boards from Annapolis Microsystems [1] which contains one to eight Xilinx 4000 series or Xilinx Vertex FPGA's as processing elements which attach to either the PCI or VME buses.

5.2 Software Developments

Several different works have been done on the development of software which can help reduce the amount of time to take a high level application and map it to a configurable computing system.

The Cameron project [11] at Colorado State University is an attempt to develop an automatic tool for image processing applications in Khoros, an advanced and widely used software development environment for signal processing. They have implemented the IP components of a standard signal-processing library called VSIPL (Vector Signal Image Processing Library) in hardware using FPGAs.

The CHAMPION project [12] at the University of Tennessee focuses on providing tools to automate the process of mapping image processing applications in Khoros onto reconfigurable systems. The approach taken is to build a library of pre-compiled primitives that can be used as building blocks of image processing applications.

The CORDS [4] project has developed a hardware/software co-synthesis system for reconfigurable real-time distributed embedded system. It takes as input a task graph, a set of communication processors, general purpose processors, and FPGA's, and assigns the tasks to the processors, and then develops the schedule for the tasks and communication events.

The WASPP project from Annapolis Microsystems [1] combines the performance advantages of an FPGA-based computer to the floating-point arithmetic and complex algorithm execution capabilities of a DSP processor.

The RTExpress Parallel Libraries [3] tool from Integrated Sensors Inc. consist of efficient, parallel performance tuned implementations of over 200 MATLAB functions. The RTExpress Parallel libraries are written in the C language. They have been designed for execution the on embedded high performance parallel architectures.

There have been several commercial efforts to generate hardware from high-level languages. The Signal Processing Workbench (SPW) from the Alta Group of Cadence, translates from a block diagram graphical language into VHDL, and synthesizes the hardware. The COSSAP tool from Synopsys also takes a Block Diagram view of an algorithm and translates it to VHDL or Verilog. However, the levels that one has to enter the design in SPW or COSSAP is at the block diagram level with interconnection of blocks which resembles structural VHDL. The Renoir tool from Mentor Graphics Corporation lets users enter state diagrams, block diagrams, truth tables or flow charts to describe digital systems graphically and the tool generates behavioral VHDL/Verilog automatically. Subsequently the Monet tool converts the behavioral VHDL into RTL VHDL, which can be used by the Leonardo logic synthesis tool. Tools such as Compilogic from Compilogic Corporation translate from C to RTL Verilog. Tools such as C2VHDL and COSYMA perform simple transformations on simple C programs to generate behavioral VHDL code. JRS Labs also has a C to VHDL translator. Again this tool translates C to behavioral VHDL.

Our MATCH compiler project [2] differs from all of the above in that it is trying to develop an integrated compilation environment for generating code for DSP and embedded processors, as well as FPGAs, using both a library-based approach and automated generation of C code for the DSP and RTL VHDL code for the FPGAs.

6 Conclusions

In this paper we provided an overview of the MATCH project. As described in the paper, the objective of the MATCH (MATlab Compiler for Heterogeneous computing systems) compiler project is to make it easier for the users to develop efficient codes for configurable computing systems. Towards this end we are implementing and evaluating an experimental prototype of a software system that will take MATLAB descriptions of various embedded systems applications in signal and image processing, and automatically map them on to an adaptive computing environment consisting of field-programmable gate arrays, embedded processors and digital signal processors built from commercial off-the-shelf components.

In this paper, we first provided an overview of the testbed which is being used to demonstrate our ideas of the MATCH compiler. We next described the various components of the MATCH compiler. Subsequently we presented preliminary experimental results of our compiler on small benchmark MATLAB programs, and compared our work with other related research.

References

[1] Annapolis Microsystems Inc., "Overview of the WILDFORCE, WILDCHILD and WILDSTAR Reconfigurable Computing Engines," http://www.annapmicro.com.

[2] P. Banerjee, A. Choudhary, S. Hauck, N. Shenoy, "The MATCH Project: MATLAB Compilation Environment for Adaptive Computing Systems." http://www.ece.nwu.edu/cpdc/Match/Match.html

[3] M. Benincasa, R. Besler, D. Brassaw, R. L. Kohler, Jr. "Rapid Development of Real-Time Systems Using RTExpress". *Proceedings of the 12th International Parallel Processing Symposium*, pages 594-599, Mar 30 - Apr 3, 1998.

[4] R. P. Dick, N. K. Jha. "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems". *IEEE/ACM International Conference on Computer Aided Design*, pages 62-68, San Jose, California, November, 1998.

[5] IEEE Computer Society, *IEEE Symp. on FPGAs for Custom Computing Machines*, Proc. of an annual conference held in Napa Valley in April, 1993-1999. http://www.fccm.org

[6] J. L. Gaudiot and L. Lombardi, "Guest Editors Introduction to Special Section on Configurable Computing," *IEEE Transactions on Computers*, Volume 48, No. 6, June 1999, pp. 553-555.

[7] T. Gaverick et al. Napa 1000. *http://www.national.com/appinfo/milaero/napa1000/*,1999

[8] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao. "The Chimaera Reconfigurable Functional Unit". *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 16-18, 1997.

[9] J. Wawrzynek, "The BRASS Research Project", http://http.cs.berkeley.edu/projects/brass/

[10] A. Agarwal et al, "Baring it all to Software: Raw Machines" *IEEE Computer*, September 1997, Pages 86-93.

[11] W. Najjar et al, "Cameron Project: High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines," *PACT Workshop on Reconfigurable Computing,* October 1998, http://www.cs.colostate.edu/cameron/

[12] D. Bouldin et al, "CHAMPION: A Software Design Environment for Adaptive Computing Systems" http://microsys6.engr.utk.edu/ bouldin/darpa/