Match Virtual Machine : An Adaptive Runtime System to execute MATLAB in Parallel

Malay Haldar, Anshuman Nayak, Abhay Kanhere,
Pramod Joisha, Nagaraj Shenoy, Alok Choudhary
and Prithviraj Banerjee

Technical Report No.CPDC-TR-2000-05-006

25th May 2000

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Abstract

MATLAB is one of the most popular languages for desktop numerical computations as well as for signal and image processing applications. Applying parallel processing techniques to improve performance of MATLAB codes has been the goal of many recent works. Most current frameworks require the user to specify parallelism and/or information regarding type/shape of the variables, thereby sacrificing the user friendliness which is one of the most popular MATLAB features. Other systems work on a restricted subset of MATLAB, thereby limiting the class of applications MATLAB can support. We present a runtime system capable of executing MATLAB code in parallel without any user intervention. The runtime system performs automatic parallelization and type/shape inference of the code at runtime. A unique feature of the runtime system is its capability to automatically adapt to changes in the underlying architecture, making it particularly useful for systems where predicting performance statically is difficult. We present experimental results obtained for the runtime system running on SGI Origin2000 shared memory multiprocessor.

1 Introduction

With $400,000$ users in over 100 countries, MATLAB is arguably one of the most popular languages for desktop numerical computing as well as image and signal processing. MATLAB is a high level language supporting real and complex matrices as basic datatypes. Thus a few line of MATLAB code can substitute hundred lines of C or Fortran codes. Along with this, the user friendly syntax of MATLAB and high quality of underlying numerical libraries have contributed much to the popularity of MATLAB. Because of the computationally intensive nature of most numerical algorithms, MATLAB seems to be an ideal target for parallelization. Current approaches to parallelizing MATLAB can be categorized into three basic strategies :

- *Parallelizing Libraries* ([9]) : This approach attempts to parallelize the underlying libraries used by MATLAB. The limitation of this approach is that parallelism is restricted within the individual libraries and higher parallelism present at the algorithm level (e.g. task parallelism) cannot be exploited.
- *Compiling* ([7],[4],[12],[8],[13],[14], [10],[27],[28],[15]) : This approach attempts to compile the MATLAB code to another language such as C/C++, Fortran, HPF etc. These target languages are considered more amenable to optimizations than MATLAB which is an interpreted language. This approach however suffers from complex type/shape analysis issues. Also the parallelizing strategies at compile time depend on static estimates of execution time of the functions and worst case behavior of control flow in the code. These factors may be sub-optimally handled through a compiler in situations where obtaining accurate estimates of execution times for functions is difficult.
- *Parallel Interpreter* ([17], [11], [16],[18]) : This approach involves adding user friendly parallelizing commands to the interpreter. The interpreter is modified to interpret these commands and execute functions in parallel. The shortcoming of this approach is that the user is required to know how to parallelize his MATLAB code. This may not be a feasible solution for most users.

In this paper we present a framework for executing MATLAB code in parallel without any user intervention. Thus a MATLAB code that runs on an interpreter can be run on multiple processors without any modification/addition to the code. The runtime system was developed as part of the MATCH (MATlab

Compiler for Heterogeneous adaptive computing systems) project at Northwestern University, and is called the **M**ATCH **V**irtual **M**achine (MVM). The MVM models the behavior of an out-of-order microprocessor in software. The MVM relies on runtime data dependency analysis to discover task parallelism. Data parallelism is exploited by efficient data parallel libraries. The scheduling and allocation decisions needed for mapping independent tasks on processors are taken at runtime based on estimates of execution time of functions. On one hand the MVM does not require the user to specify any directives regarding type/shape or parallelization, thus preserving the user friendly characteristics of MATLAB. On the other hand it greatly simplifies the compiler since no type/shape analysis is required at compile time. Note that the MVM represents a concept and not necessarily any particular implementation. The back end of the MVM can be tailored to run MATLAB code on a variety of parallel and heterogeneous architectures. Figure 1 gives an overview of the framework.

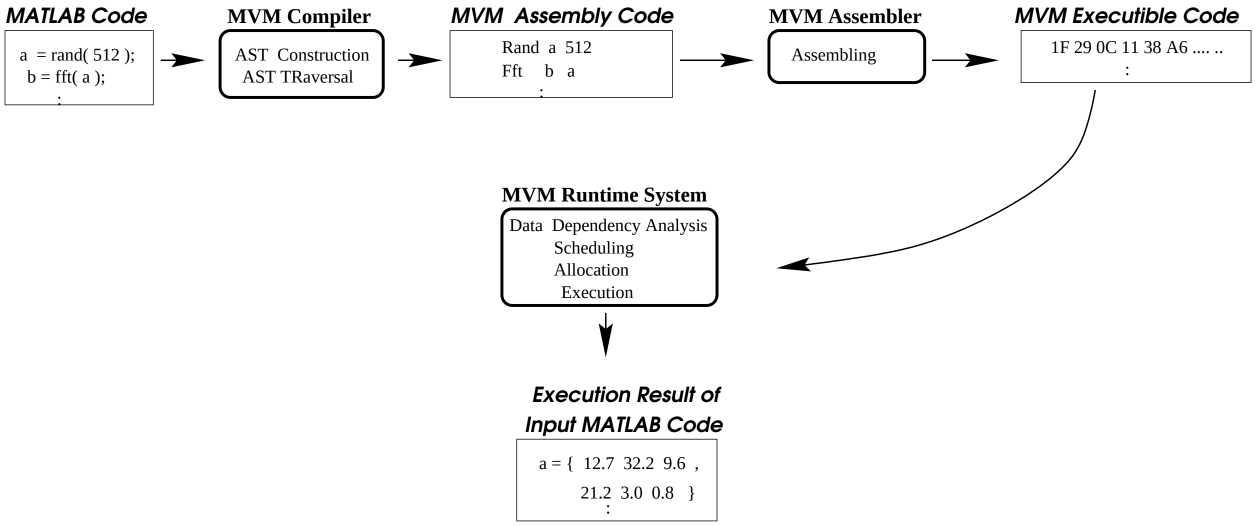

Figure 1: Overview of the Match Virtual Machine framework.

The rest of the paper is organized as follows - Section 2 describes an overview of the MVM architecture, the runtime system and an associated compiler. Section 3 describes enhancements to the basic MVM architecture to exploit more parallelism. Section 4 deals with some experimental results obtained for an implementation of the MVM on SGI Origin2000. Section 5 discusses future extensions and Section 6 presents some conclusions of the present work.

2 Overview of the MATCH Virtual Machine

In the MVM framework, the task of executing MATLAB code in parallel is broken down into two steps - **(1)** Compiling the given MATLAB code to MVM assembly instructions, followed by **(2)** Executing the MVM assembly instructions in parallel by data dependency analysis on the instructions at runtime. First we describe the architecture of the MVM which models a contemporary high performance microprocessor in software. MVM views the different components of a multiprocessor system as functional units on which functions (such as filter,fft etc) can be executed. The scheduling and allocation of the instructions are dependent on cost estimates of executing the instructions which can be modified during execution. We also describe the assembly instructions for the MVM. Next we describe a compiler that takes MATLAB code as input and compiles it to the MVM assembly, ready for execution by the MVM.

2.1 The MVM Architecture

The MVM simulates the behavior of an out-of-order microprocessor execution core. The input to the MVM is an executable written in the MVM assembly language. Each of these assembly instructions may represent functions/operations present in MATLAB (e.g. fft, * etc). In addition, there are other instructions such as control flow instructions (jump), scalar comparisons (set-less-than, set-greater-than) etc that are necessary for handling arbitrary data and control flow. The output of the MVM is the result obtained by executing the input executable code.

Typically, the MVM runs on a processor that has the ability to initiate execution of functions on other processors in the multiprocessor system. Executing an MVM assembly instruction involves initiation of execution of the function by the MVM and collecting back the results. MVM primarily relies on efficient libraries on the different components of the multiprocessor system (which may be heterogeneous) to execute the functions. However, if the computation involved in executing an instruction is very small (like scalar operations, jumps etc.), the instructions are executed locally by the MVM itself, without using the remote execution mechanism.

Figure 2 shows the MVM architecture. Conceptually, the MVM accesses two memories - the Instruction Memory and the Data Memory. The Instruction Memory is an array of integers which stores the input

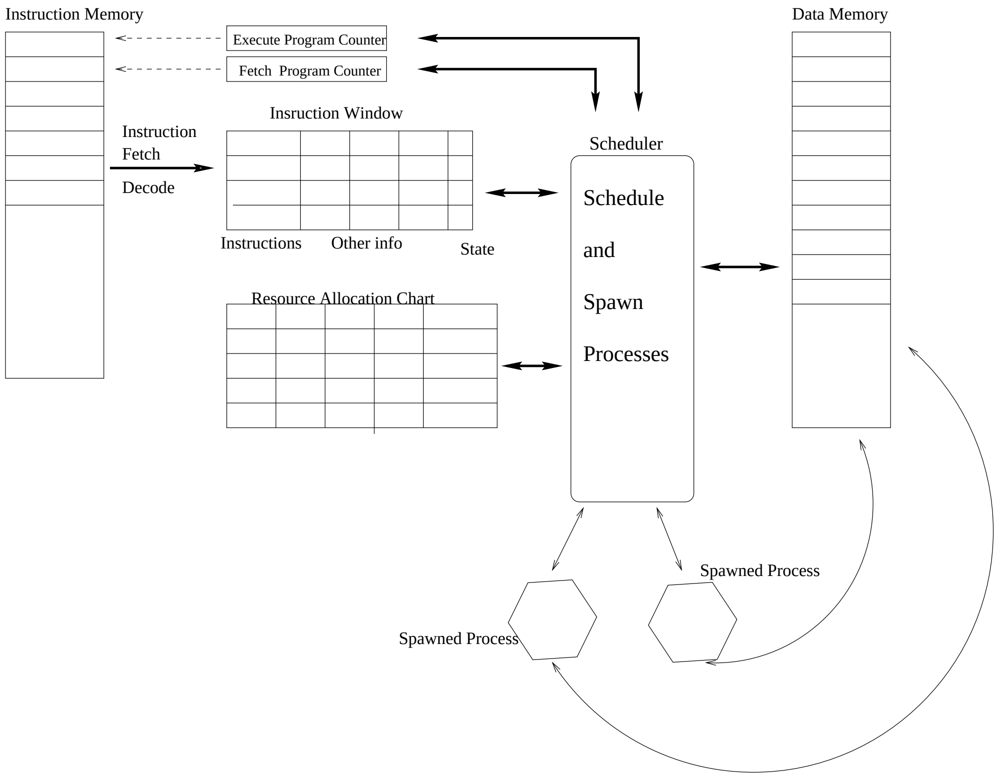

Figure 2: The MVM Architecture

executable in consecutive locations of the array. The Data Memory is an array of void pointers that may point to the results of function execution. The pointed data may be of any type/shape. The necessary type/shape information is tagged on to each entry of the Data Memory. Typically, the pointed data are matrices stored in column major format with the tags indicating their dimension, bounds, type, precision etc. Note that both the Instruction Memory and the Data Memory are abstract data structures of the MVM, and not any actual physical memory.

The MVM maintains a window of instructions that contains a set of instructions that the MVM is considering to execute at any point of time. The instruction window is filled by fetching instructions from the Instruction Memory. The location in the Instruction Memory from where the next instruction is to be fetched is pointed by the Fetch Program Counter of the MVM. In addition, the MVM has an Execute Program Counter which points to the location in the Instruction Memory until which all instructions have been successfully completed and committed. The MVM first finds the instructions within the window that have all the variables they use ready i.e, all their operands have been computed correctly by some previous instructions. Note that some instructions have no operands that they read, and therefore are always ready for execution. Once all such instructions are identified, the MVM starts the execution of these functions on available resources. The mapping of instructions to resources is dictated by the cost estimates of executing the function on different resources and the scheduling algorithm used. Results from the function execution are written into the Data Memory along with all necessary type/shape informations. Completed instructions are removed from the instruction window and new instructions are fetched. The scheduling algorithm is repeatedly applied to the window until all instructions are executed. Note that if the MVM identifies two instructions both of which have all their operands ready, and there are two resources that can execute these two instructions, then the MVM can start executing both instructions simultaneously. This is the basic mechanism by which the MVM exploits parallelism. It should be noted that this scheme is suitable to exploit only *task parallelism*. To exploit *data parallelism* MVM relies on libraries written is a data-parallel manner. Certain concepts like jump prediction, register renaming etc have been borrowed from typical high performance microprocessor architectures to further improve the performance of the MVM. These are described in Section 2.5. Execution on the MVM is discussed in detail in [1]. A brief summary of the steps of execution are as follows :

- *Fill Window* : This step attempts to fill the instruction window by fetching instructions from the Instruction Memory. If a branch instruction is encountered then the outcome of the branch is predicted (Section 3.1). The fetched operands are renamed (Section 3.2).
- *Schedule* : This step involves finding the instructions in the instruction window that have all the variables they use already computed. These instructions are marked “ready” and are considered for allocation.
- *Allocate* : The set of ready instructions are mapped onto the set of available resources for execution. Execution of the mapped functions are initiated after mapping. A resource becomes unavailable once a function has been assigned to it. It becomes available after the completion of the function.
- *Commit* : An instruction is committed i.e, the Execution Program Counter of the MVM is advanced beyond the instruction when the execution of the instruction is complete and all prior instructions have

been committed.

The above steps are repeated until the execution of the input to the MVM is complete. Thus, the MVM fetches instructions and commits them *in-order*. However, the execution and completion of the instructions may be *out-of-order*.

2.2 Allocation Algorithms

As mentioned in Section 2.1, after instructions ready for execution have been identified, the instructions are mapped onto resources based on the their expected time to completion and the allocation algorithm used. Three different allocation algorithms have been implemented in the MVM. The purpose here is not to suggest a best allocation algorithm for the MVM framework, but to investigate the effect of different allocation algorithms. We show in Section 4.3 some of the issues while comparing the different allocation algorithms implemented. The three allocation algorithms implemented are :

- **Arbitrary** : Given a set of instructions ready to execute and a set of resources capable of executing these instructions, the arbitrary scheduling algorithm assigns instructions to the resources without taking into account any cost metric. This algorithm is extremely fast and does not require any kind of cost estimation. However, given heterogeneity of resources and different computational needs of different functions, arbitrary scheduling algorithm may not be the best choice.

- **Shortest Job First** : The shortest job first (SJF) algorithm finds the instruction that takes the minimum time among all ready instructions considered over all available resources. The instruction is then mapped to the corresponding resource.

- **Longest Job First** : The longest job first (LJF) algorithm finds the instruction who's smallest expected execution time is largest among all ready instructions considered over all available resources.

The mapping problem with the objective of finishing all the functions in least time possible has been proven to be NP complete [21]. Typically, greedy heuristics are used to solve the scheduling problem where schedules are produced at runtime [19], [20], [22].

2.3 Cost Estimation

The allocation algorithms described in Section 2.2 depend on the cost estimates for the instructions. Typically these costs are expected time to complete. Since in MVM the scheduling decisions are taken at runtime, the cost estimates can be modified and hence adapted during runtime. The cost estimation algorithm takes the function, the operand sizes and resource as input and returns an estimate of the time it will take to complete the function on the resource. The algorithm can take into consideration the validity of its earlier predictions and can modify accordingly. Also it is easy to extend the cost estimation algorithm to take into account various other factors affecting the execution time of the functions. For example, if the computational resources consists of a cluster of workstation shared by users, the time to execute a function on a resource depends on the current load on the workstation. The cost estimation algorithm can be modified to take into account these factors to give better estimations, which results in a more meaningful schedule. Note that initial estimates are provided by the MVM based on some static estimates. A typical compiler would use these static estimates for scheduling the complete program. The MVM however, can modify these estimates as the execution proceeds.

2.4 The MVM Assembly Instructions

This section describes the format for the MVM assembly instructions. As mentioned earlier, these instructions represent functions present in MATLAB and some scalar arithmetic operations and control flow instructions for the MVM. Two main considerations in the design are - **(1)** Fast decoding and **(2)** Easy extension. Fast decoding is crucial to have low overhead of running the MVM. Also the format should be such that new assembly instructions can be easily added.

MVM assembly instructions are of variable lengths (in words) and the format is shown in Figure 3. Each word is 4 bytes long. The variable length format is chosen for easy extensibility and considering the wide range of the number of operands required for different MATLAB functions. For fast decoding, critical information regarding the instruction is encoded in a fixed format in the first word of the instruction. The first word of all the instructions is reserved for the opcode. The operands and immediate values follow the opcode. The number of operands is dependent on the instruction. The lower 8 bits of the opcode indicate the number of operands. Therefore maximum number of operands is limited to 255. The upper 8 bits are 0s for all non-control-flow instructions. For control flow instructions the upper 8 bits denote the type of control flow instruction. Hence, the kind of control flow instructions is limited to 255. For non-control-flow instructions, the middle 16 bits indicate the type of operation. Hence total number of non-control-flow instructions is limited to 65535. A list of opcodes implemented in the current version of the MVM can be found in [1].

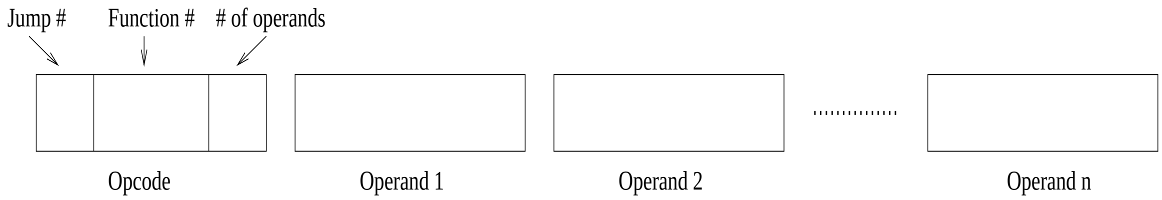

Figure 3: MVM Assembly Format

Following the opcode are the operands. The operands are integers and are taken as locations of the Data Memory array. For example, the integer 5 in the operand field denotes that the operand is the data pointed by the 5th entry of the Data Memory array. The operand of certain instructions are treated as immediate values. Note that in the assembly instructions no information regarding the type/shape of the operands are encoded. Whenever necessary, special instructions for scalar only operands are constructed to avoid overheads.

2.5 Compiling to the MVM

One of the main advantages of the MVM is the ease of compilation to it. The compilation process just transforms the MATLAB code to MVM assembly instructions that is simple to handle at runtime and does not add any information or optimizations. The first step involved in compilation is parsing the input MATLAB program based on a formal grammar and building an abstract syntax tree. The grammar used for MATLAB 5.2 is given in [6]. As the code compiled for the MVM does not have type/shape information, typical type shape analysis methods required at compile time are not needed. This reduces the complexity of the compiler to a great extent. Also, the MVM makes the parallelization and scheduling decisions automatically at runtime. Hence typical parallelization annotations required in other frameworks are also not needed and the user is not required to know parallelizing strategies. A simple traversal of the abstract syntax tree is used to produce the code. It is assumed that for all operations and functions present in the input MATLAB program, there exists at least one library function on one of the components of the multiprocessor system. Exceptions are scalar operations which have corresponding operators in C. Figure 4(a) shows a code generation example.

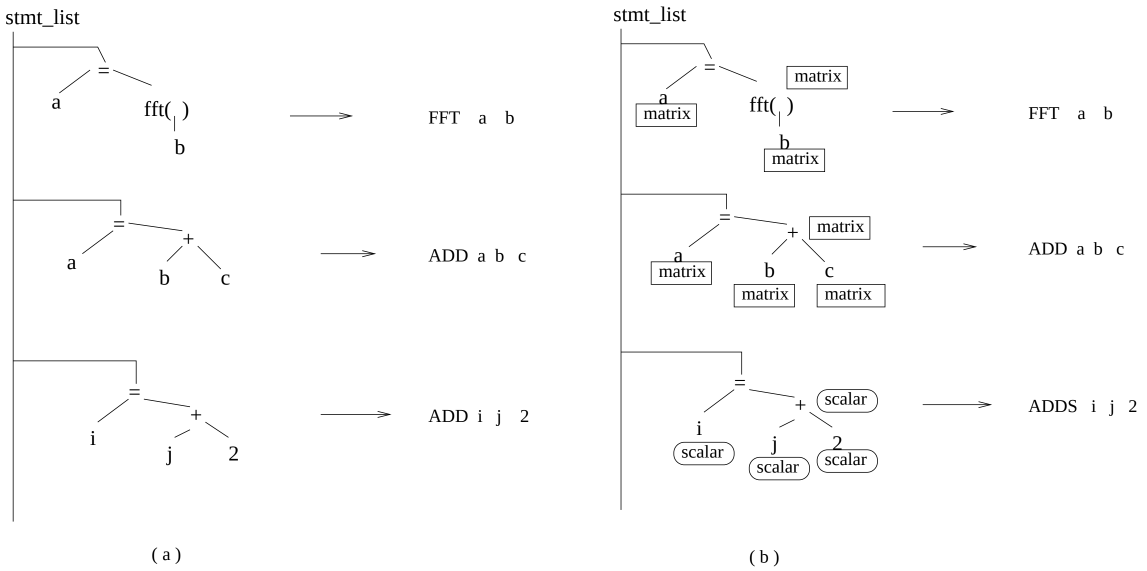

Figure 4: Code Generation for the MVM:(a) A simple postorder traversal of the AST produces an executable consisting of assembly instructions for the MVM. The generated code does not have type/shape information. (b) Optimizations regarding scalars can be done by inferencing the scalar operations and generating instructions which handle scalars only.

Note that although the addition for *a* and *b* is matrix addition, whereas the addition for *j* and *2* is scalar addition, the generated code is *ADD* for both the cases. However, this may result in unacceptable overheads for executing scalar operations. To avoid this problem, there are instructions in the MVM that operate solely on scalar operands. Also by a simple type promotion algorithm, wherein the output of an operation is marked scalar only if operands are scalar (except for certain operations), most of the scalar operations can be found in the input program. This information regarding whether an operation is guaranteed to have scalar operands or not can be annotated on the AST. While generating assembly code, if operations are annotated to be scalar, then the low overhead scalar only assembly instructions can be generated. The scalar only assembly instructions have low overheads as they do not invoke the remote execution mechanism. Figure

4(b) shows the modified example.

3 Exposing more Parallelism in the MVM

3.1 Control Flow Prediction

To spot opportunities for parallel execution, the instruction window must be nearly full most of the time. The greater the number of instructions, greater are the chances of discovering instructions that can be executed in parallel. However, control flow instructions such as jump disrupt the sequence in which instructions are to be fetched. While fetching, if we encounter a jump instruction it is not clear whether we should fetch the next instruction or the instruction that is the target of the jump instruction. To solve this problem we use a classical control flow prediction mechanism used in microprocessor architectures [3]. Control flow prediction is also referred to as branch prediction or jump prediction. Outcome of a control flow instruction is characterized as "taken", implying the next instruction to be executed is not the instruction following the control flow instruction, but has to be fetched from the "target" of the control flow instruction. A "not taken" outcome of a control flow instruction implies that the instruction flow is not disrupted by the control flow instruction and the next instruction to be executed in the instruction that appears after the control flow instruction. The control flow prediction mechanism consists of a control flow prediction table. Each entry of the table has a 2 bit saturating counter. The table is accessed by using the lower bits of the Fetch Program Counter of the MVM as index. If a particular entry is below 2, the prediction is supposed to be taken, else the prediction is the control flow will not be taken. This allows us to fill the instruction window speculatively. If the control flow happens to be mispredicted, then all instructions following the control flow instruction are invalidated and a new fetch is initiated depending upon the result of the control flow instruction. After executing a control flow instruction, the corresponding entry of the control flow prediction table is modified. If the outcome of the control flow was taken, then the counter is decremented, else incremented.

Thus while filling the instruction window, the MVM fetches instructions from the Instruction Memory. If a control flow instruction is fetched then the next instruction is fetched depending upon the prediction from the control flow prediction table and the target of the control flow instruction instruction.

3.2 Variable Renaming

Data dependencies can be introduced in the program due to variables having the same name. Write after Read (WAR) and Write after Write (WAW) hazards can be eliminated by renaming the variables such that each variable is defined only once. This is particularly important because of the fact that control flow prediction results in unrolling of loops wherein a lot of variables appear with same names. To fully exploit parallelism, true data dependencies must be distinguished from WAR and WAW dependencies. The MVM maintains a mapping of variable names to actual Data Memory indices. When an instruction is fetched, first the operands that are used by the instructions are replaced with their corresponding new names. Then the variables that the instruction defines is given a new name and a corresponding entry is made in the renaming table. Ideally, the memory allocated for the renamed variable should be freed after the execution of the last instruction using the variable is over. However, directly detecting the last instruction to reference a variable at runtime is complicated. An easier alternative is to free the renamed variable corresponding to a variable say x, when the next renamed variable for x is committed, since beyond that point no reference to the old name of x can appear. Figure 5 shows an example of variable renaming.

3.3 Misprediction Correction

As mentioned earlier, when control flow instructions are encountered during the fetch stage, they are predicted. At a later stage the control flow instructions are actually executed. If the result of the execution of the control flow instruction matches its prediction, it is simply ignored. However, if the execution outcome does not match the earlier prediction, then the state of the virtual machine must be restored to the state before the branch was fetched. This is done in two steps :

- First, the names of the variables must be restored to the point where the control flow instruction was fetched. For this an iteration is executed starting from the end of the instruction window and ending at the entry where the mis-predicted control flow instruction is there. At each step of the iteration, the names of the renamed variables are restored back to their old names. The old names are stored along with each location of the Data Memory array to facilitate this.

- After restoring the names, all the instructions following the mispredicted control flow instruction are

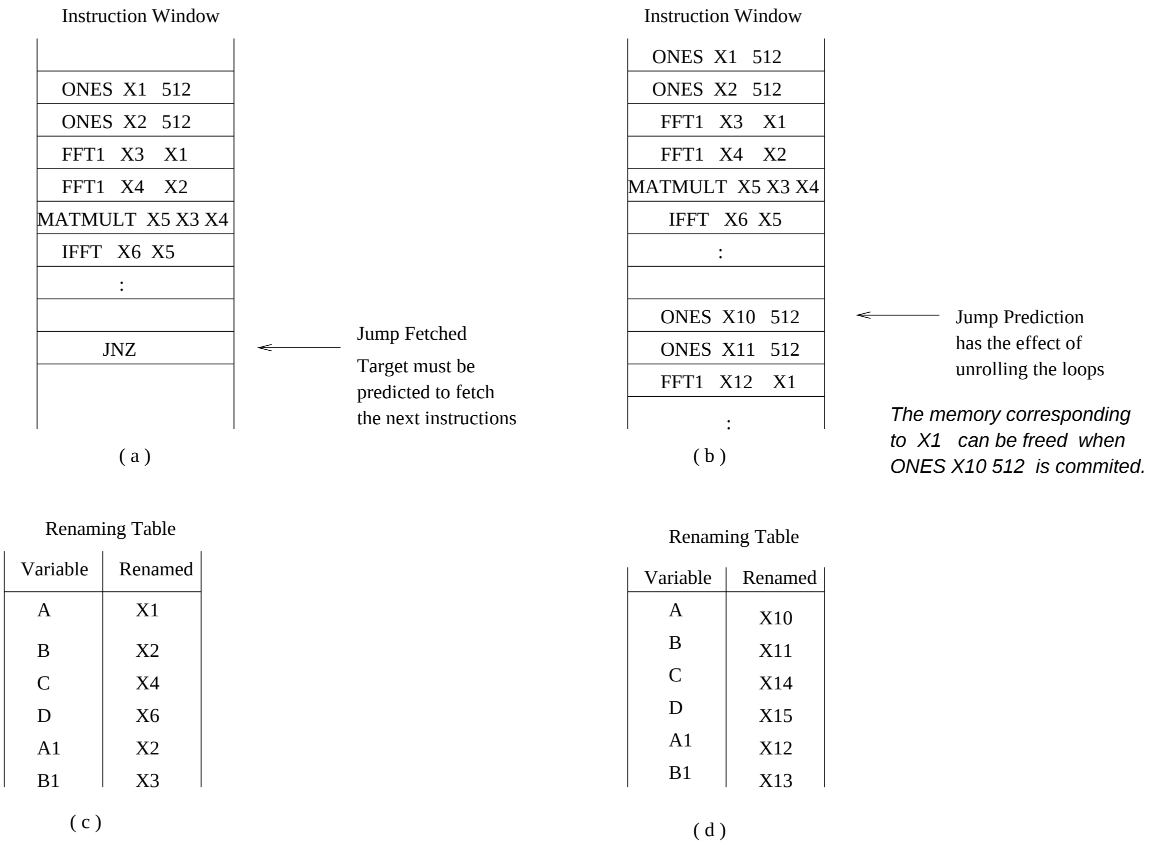

Figure 5: Example of Variable Renaming: (a) shows the instruction window before predicting the jump. (b) shows the renaming table prior to predicting the jump. After JNZ is predicted the instructions for the next iteration of the loop are fetched. The instructions have same name of the operands across the loop iterations. However, to effectively exploit parallelism, each variable is given a unique name at definition. Thus, after the jump is predicted and ONES fetched, the name of A is changed from $X1$ to $X10$. (c) shows the instruction window after the jump, and (d) shows the renaming table after the jump.

flushed. If execution of some of the flushed instructions were started, then their results are simply ignored. In the current implementation of the MVM, no mechanism is provided to stop the execution of instructions that lie in the mispredicted path, once they are started. After flushing, new instructions are fetched from the corrected target of the branch.

4 Experimental Results

4.1 Experimental Environments

The results presented in this section are mostly for the MVM running on an 8 processor SGI Origin2000 shared memory multiprocessor. Each of the eight processors is a 64-bit MIPS R10000 225MHz processor with 32Kb of primary cache. Results for the MVM running on a Sun workstation is also included. The workstation had a single 269MHz UltraSPARC-IIi processor running SunOS 5.6. The benchmarks used include the image correlation benchmark (Figure 6) and synthetically generated benchmarks. The outer loop of the image correlation benchmark simulates the case where we are correlating a stream of images. The synthetic benchmarks were generated by a tool that can be used to generate MATLAB codes with desired characteristics. The tool takes dependent steps, function mix, parallelism, size of the metrics etc as expected statistical averages and produces MATLAB codes having those characteristics. The principle objective of the experiments was to demonstrate that if the input codes had parallelism, then the MVM framework's ability to exploit the parallelism was comparable to that of hand optimized versions.

for $i =$ *1:10*
a=rand(512);
b=rand(512);
a1=fft(a);
b1=fft(b);
c1=a1*b1;
d =ifft(c1);
end;
	ASSIGNS iters 10
	ASSIGNS loops 0
	ASSIGNS one 1
	>> for_loop1 RAND a 512
	RAND b 512
	FFT1 a1 A
	FFT1 b1 B
	MATMULT c a1 b1
	IFFT d c
	ADDS loops loops one
	SUBS over iters loops
	JNZ over for_loop1
	END

Figure 6: The Image Correlation benchmark and the corresponding MVM assembly code generated.

4.2 Instruction Window Size

Figure 7 shows the variation in execution time with varying instruction window sizes and number of processors for the image correlation benchmark using arbitrary scheduling. As seen the speedups saturate beyond 3 processors for an instruction window size of 32 instructions. However, for an instruction window of 64 instructions, we continue to get speedups till 7 processors. This confirms the fact that a larger instruction window helps in exposing more parallelism, and we are able to employ more processors usefully to get better speedups.

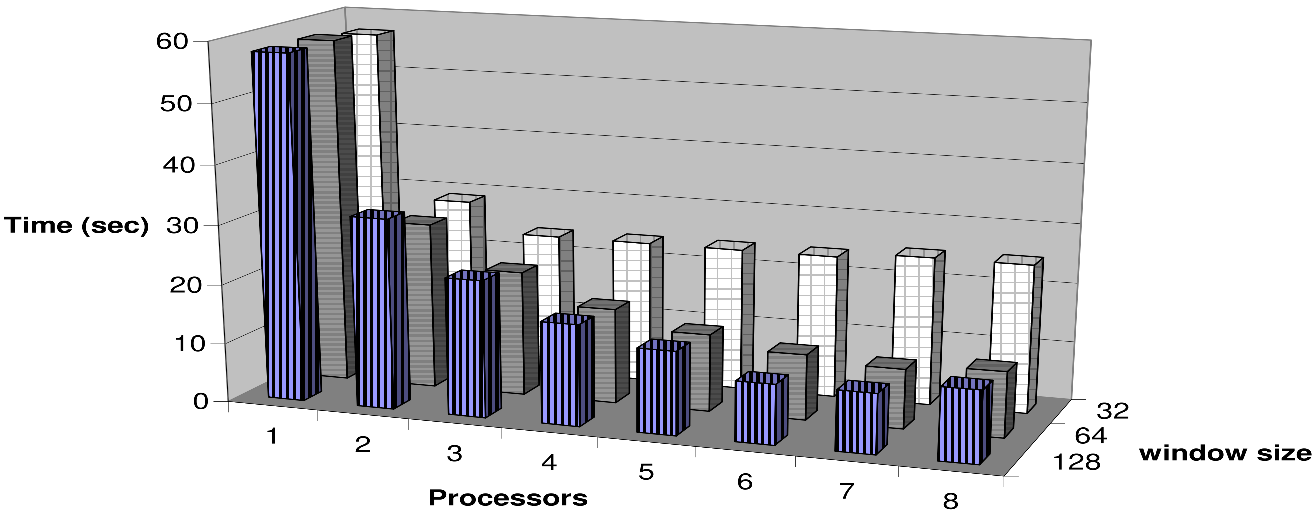

Figure 7: Execution times for Image Correlation benchmark with arbitrary scheduling. Variation with number of processors (1 to 8) and instruction window size $(32, 64$ and $128)$. Times are in seconds

4.3 Scheduling Algorithms

Figure 7, Figure 8 and Figure 9 show the variation in execution time for the arbitrary, SJF and the LJF scheduling algorithms respectively. As can be seen, the execution time for the LJF algorithm is significantly higher than the SJF or the arbitrary scheduling algorithm for the same number of processors and instruction window size. This can be related to the fact that LJF scheduling increases the cost of misprediction dramatically. Since branch instructions have very small costs when compared to matrix operations, the execution of branch instructions receive less priority in LJF scheduling. This delays the execution of branch instructions as other matrix operations get precedence. For a mispredicted branch, a lot of matrix operations in the wrong control path gets executed before the mispredicted branch is executed and corrected. This increases the cost of misprediction substantially when compared to SJF or Arbitrary scheduling. Also the situation gets worse with increase in the instruction window size as can be seen from Figure 9.

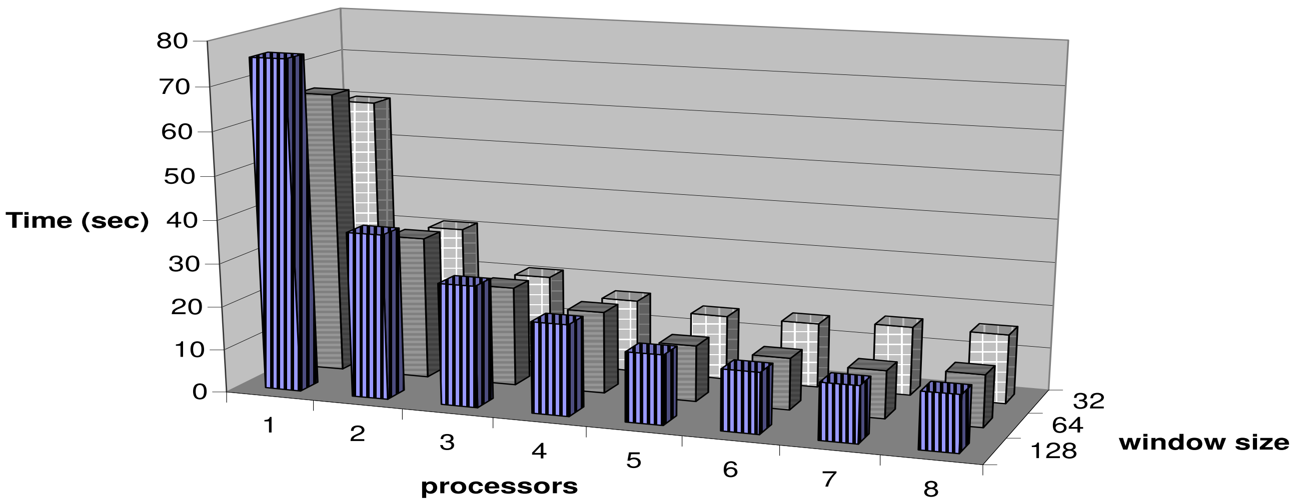

Figure 8: Execution times for Image Correlation benchmark with SJF scheduling. Variation with number of processors(1 to 8) and instruction window size (32, 64 and 128). Times are in seconds.

A corrective measure is to introduce scalar and branch operations with very large costs while using LJF scheduling. These large cost scalar and branch operations are used by the compiler while compiling control flow constructs such as loops. This gives priority to control flow instructions and reduces the misprediction penalty for LJF scheduling. For scalar operations that are not related to control flow, scalar operations with normal costs are used. Figure 10 shows the execution time with this corrective measure.

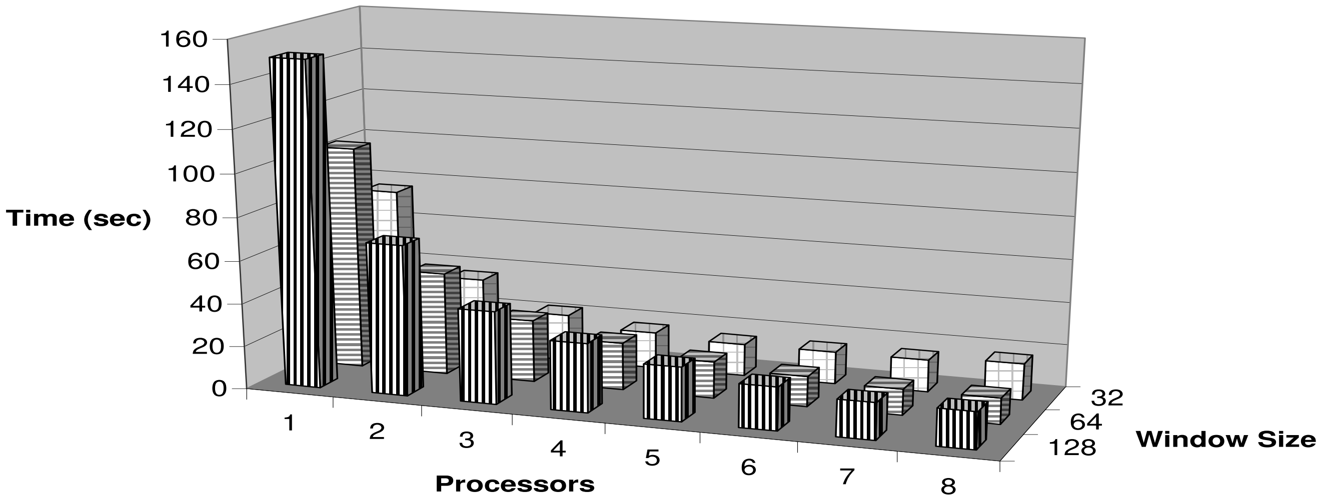

Figure 9: Execution times for Image Correlation benchmark with LJF scheduling. Variation with number of processors (1 to 8) and instruction window size (32, 64 and 128). Times are in seconds.

4.4 Branch Prediction

To illustrate the effectiveness of branch prediction mechanism we choose the code shown in Figure 11. Such loops are typical in iterative solvers.

Figure 11 shows the code that is run with and without the branch prediction mechanism. Intuitively, without branch prediction, we can exploit the parallelism present within a basic block only. This may severely limit the parallelism available if the basic blocks are small and even increasing window sizes will be unable to uncover any more parallelism. Most of the performance comes by predicting branches that capture the loops in the input MATLAB code. Predicting these branches has the effect of *unrolling* these loops, exposing more parallelism. Table 1 shows the execution times with and without the branch prediction mechanism. The *wbp* column is for "with branch prediction" and *wobp* column is for "without branch prediction". As seen, neither increasing the number of processors nor increasing the window size improves performance in the

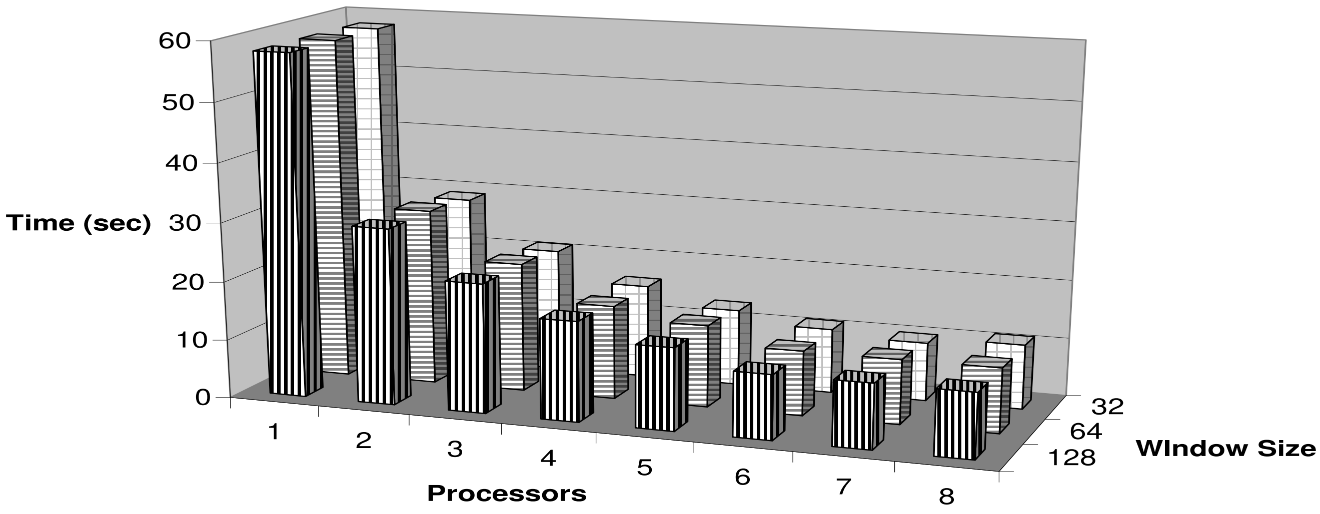

Figure 10: Execution time for Image Correlation with LJF scheduling and modified costs. Variation with number of processors (1 to 8) and instruction window size ($32, 64$ and 128). Times are in seconds.

absence of branch prediction. Whereas with branch prediction, these parameters show considerable increase in performance. Note that we have chosen the code particularly to emphasize the importance of branch prediction.

4.5 Data Parallelism

Figure 13 shows the execution times for different window sizes and number of processors for the synthetic benchmarks. The synthetic benchmarks has a mix of matrix multiplication, addition, and negation. Each execution time is average of 15 benchmarks. As shown the speedups saturate quickly beyond 3 processors.

Close analysis showed that the execution time was lower bounded by the number of multiplications to be executed serially and the time taken for the multiplication. This brings out an important limitation of using only task parallelism. The runtime is dominated by compute intensive functions (like multiplication) and relying only on task parallelism for functions with different complexities (like multiplication and addition)

```
mat1=rand(512);
mat2=rand(512);
while(mat1 > mat2 )
    a1 = rand(512);
    a2 = rand(512);
    mat1 = a1*a2;
    a1 = rand(512);
    a2 = rand(512);
    mat2 = a1*a2;
end;
```

Figure 11: MATLAB code to emphasize the branch prediction scheme.

Table 1: Image Correlation on MVM running on Origin2k with branch prediction (wbp) and without branch prediction (wobp). The scheduling algorithm is arbitrary. Times are in seconds. Figure 12 shows a graphical view of the data.

	Number of processors													
Window	1		2		3		4		5		6		7	
size	wbp	wobp	wbp	wobp	wbp	wobp	wbp	wobp	wbp	wobp	wbp	wobp	wbp	wobp
32	33.04	35.01	16.63	17.01	13.41	16.16	10.92	18.54	7.69	18.65	6.87	16.30	7.51	17.82
64	33.88	35.22	17.48	17.05	13.80	16.17	13.34	17.38	7.04	17.88	6.95	17.50	7.01	17.67
128	35.01	35.20	20.20	17.25	13.08	17.01	10.14	17.95	7.89	18.23	6.83	17.29	6.99	17.11

produces load imbalance and suboptimal resource utilization. The solution to the problem is to use data parallelism as well. The MVM exploits data parallelism by calling libraries for functions written in a data parallel manner. Introduction of data parallelism complicates scheduling decisions as the possible mappings of functions to available resources rise exponentially when compared to the only task parallel scheduling case, which itself is NP complete. At any point, the MVM distributes the available resources equally among the remaining tasks to be completed. This results in near optimal load balancing. Figure 14 gives an illustration.

As can be seen in Figure 15, we obtain linear speedups across all window sizes and number of processors.

4.6 Overheads

In this section we address the concerns regarding the overhead the MVM introduces. Table 2 shows the execution times of 4 synthetic benchmarks on MVM compared with the best manual approach. The best manual approach involves finding an optimal schedule for the benchmark and hand coding a parallel C program for it. The overheads are about 15%.

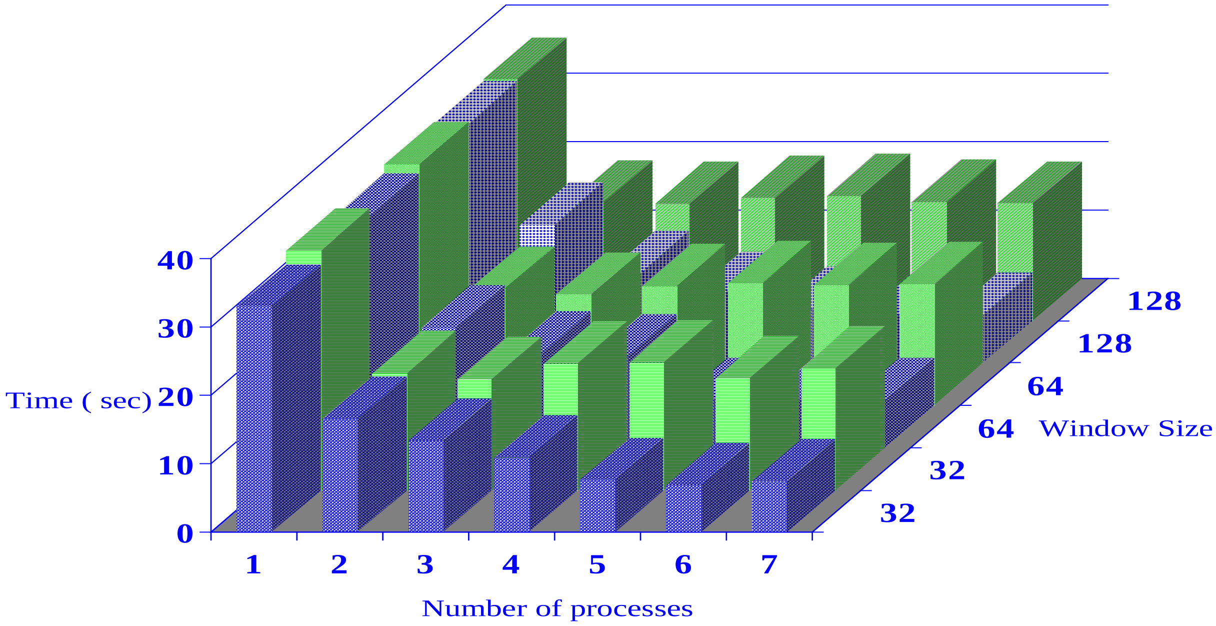

Figure 12: Performance of MVM with and without branch prediction.

In Table 3 we show the execution time for executing the code shown in Figure 16, for increasing values of N. The results are for the MVM running on an Origin 2000 against compiled C code. As can be seen, beyond matrices of size 256, the performance of MVM approaches the performance of an equivalent compiled C code. Combined with the fact observed previously that the MVM produces near optimal parallelization, we conclude that the overheads associated with MVM is quite low. In particular, the MVM can give performance within 10% of the best manual approach. For the image correlation benchmark the best manual approach achieved an execution time of $26.86 secs$, whereas the MVM achieved an execution time of $28.12 secs$. Note that the manual approach involves detailed analysis of the algorithm and tedious programming effort, whereas the MVM does not require anything beyond the MATLAB description of the algorithm.

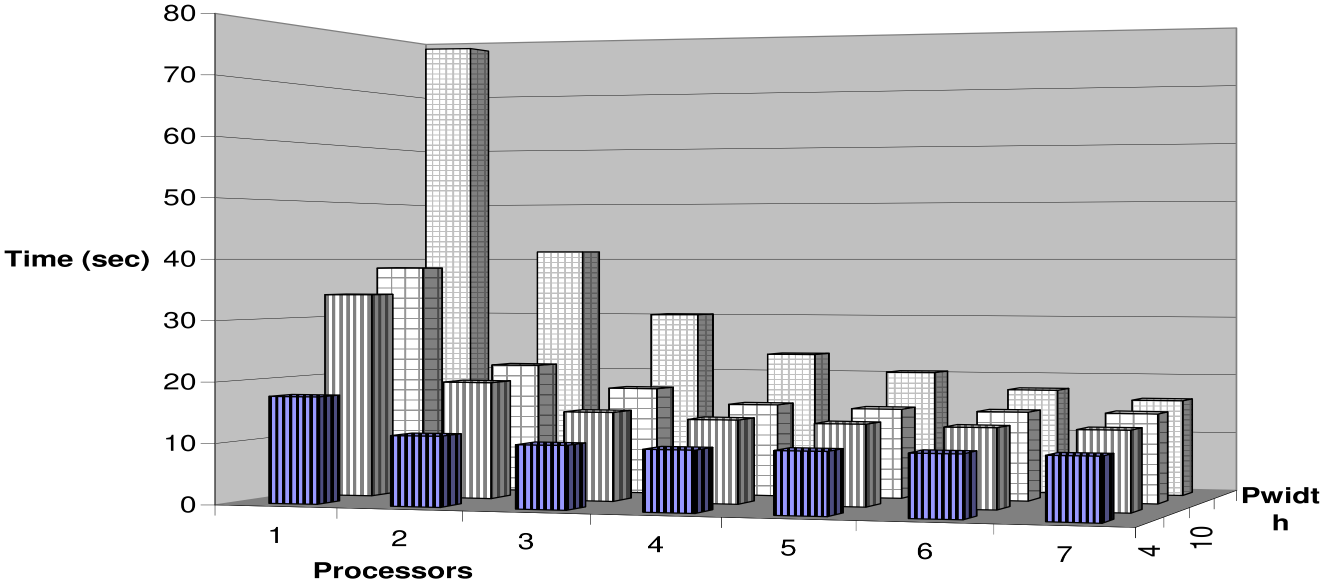

Figure 13: Execution times of synthetic benchmarks.

5 Future Work

5.1 Cluster Computing

With falling latencies and growing bandwidth of networks, clusters of computers have become viable alternatives to super computers of the past. Also these clusters are cost effective and widely available. Such a cluster may be constructed by utilizing the idle time of workstations. Projects like the Condor [24], the U. C. Berkeley NOW [25], Beowulf [26], etc., provide a wealth of information. In such a scenario, the MVM design seems very attractive. The fact that the MVM makes all the scheduling and allocation decisions at runtime enables it to handle varying number of computational resources with varying loads. In the case where resources become unavailable, all that needs to be done is that the resource has to be marked unavailable. When the resource becomes available the flag can be turned on to indicate the same. No other changes are necessary. This idea can be extended to handle fault tolerance where some of the resources can go down while executing an instruction. In such a case, the instruction can be restarted at a different

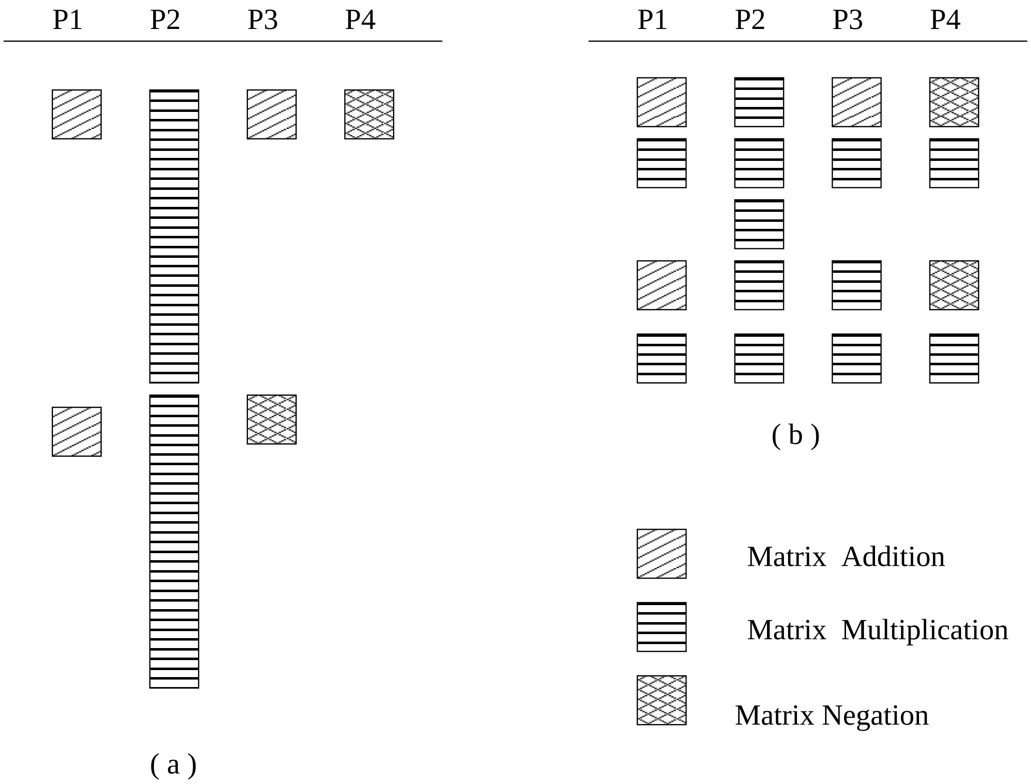

Figure 14: Effectiveness of Data Parallelism: (a) Only using task parallelism, execution time bound by serial multiplications. (b) Using both task and data parallelism, near optimal load balance.

resource after cleaning up the threads spawned earlier to execute the instruction. Handling all these issues through a compiler is more complex.

5.2 Scheduling Algorithms

We have implemented a few scheduling algorithms in the MVM scheme. The performance and applicability of other scheduling algorithms present in the literature ([19], [20], [22]) is an interesting research issue. The scheduling algorithms presented in this paper view the time to spawn a function on a remote platform, execute the function and get back the result as one composite time. Separating the different components and designing scheduling algorithms to optimize performance by overlapping the different components is an interesting problem. Along this line, a further optimization is to avoid collecting back intermediate results after the completion of each function, and do a series of operations before collecting the result back. Reconfigurable components are predicted to be an important part of future heterogeneous systems. Designing

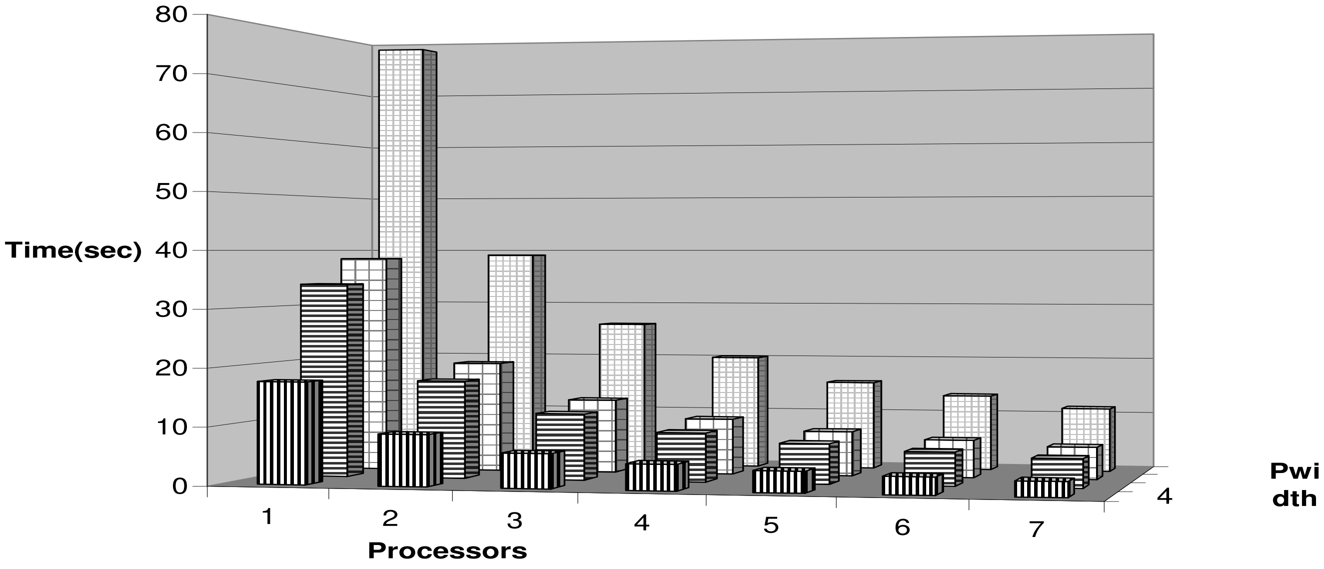

Figure 15: Execution times of synthetic benchmarks with data parallelism.

scheduling techniques to exploit reconfigurability effectively is another interesting issue.

5.3 Estimation Techniques

The MVM provides a framework to experiment with a variety of cost estimation techniques. Currently substantial literature exists on obtaining costs of functions [23]. Investigating techniques to dynamically find the cost estimates is an interesting problem.

6 Conclusions

We have presented a new framework to execute MATLAB in parallel. The framework involves compiling MATLAB to an abstract machine (called the MATCH Virtual Machine) that does runtime dependency analysis and allocation, much like a modern high performance microprocessor. The framework does not require any user intervention to execute the MATLAB code in parallel. Since most of the analysis and

Table 2: Comparison of execution times for 4 synthetic benchmarks. The execution times on the MVM are compared against hand coded C versions of the benchmarks with optimal scheduling. Times are in seconds.

	Benchmarks			
Platforms	Synth1	Synth2	Synth3	Synth4
C Code	1.43	1.21	1.26	1.35
MVM	1.67	1.27	1.43	1.55
Overhead	17%	5%	13%	16%

```
tic;
N=4;
a=ones(N);
b=ones(N);
c=a*b;
toc;
```

Figure 16: MATLAB code to determine the MVM overhead.

allocation is done at runtime, the compiler is highly simplified and no compile time type/shape analysis or data dependency analysis is needed. The framework is capable of exploiting both task and data parallelism and can adapt to changes in the characteristics of the underlying multiprocessor system it is running on. An implementation of the framework on SGI Origin2000 shared memory multiprocessor indicate that performance within 15% of hand optimized and parallelized code can be obtained. The present work also opens interesting research avenues in investigating the adaptation of the framework to different multiprocessor systems and in designing new scheduling/allocation strategies that the framework demands.

References

[1] M. Haldar, *A Library based MATLAB Compiler and Runtime System for Adaptive Heterogeneous Platforms,* M.S Thesis, Northwestern University, Dec. 1999.

[2] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, Y. A. Li, *Heterogeneous Computing*, "Handbook of Parallel and Distributed Computing", A. Y. Zomaya Editor, pp. 725-761, McGraw-Hill, New York, NY, 1996.

[3] J. L. Hennessy, D. A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers,Inc., 1996.

[4] P. Banerjee et al, *MATCH: A MATLAB Compiler for Configurable Computing Systems*, Technical Report, Center for Parallel and Distributed Computing, Northwestern University, Aug. 1999, CPDC-TR-9908-013.

[5] P. Joisha, A. Kanhere, U. Shenoy, A. Choudhary, and P. Banerjee, *An Algebraic Framework for Array Shape Inferencing in MATLAB* , Submitted to Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).Also appears as Technical Report, Center for Parallel and Dist. Computing, Northwestern University, CPDC-99-11-019, June 2000.

[6] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U.Nagaraj Shenoy and Alok Choudhary, *The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler* , Technical Report, Center for Parallel and Distributed Computing, Northwestern University, CPDC-TR-9909-017, Sep. 1999.

[7] L. DeRose and D. Padua, *A MATLAB to Fortran90 Translator and its Effectiveness*, Proc. 10th ACM Int. Conf. Supercomputing (ICS), May 1996.

Table 3: Overhead of the MVM : Comparison of execution times of code shown in Figure 16 when run of the MVM with 1 processor against compiled C code, for increasing matrix sizes. Times are in seconds.

	Matrix Size (N)						
Platforms	16	32	64	128	256	512	1024
Compiled C Code	0.00	0.00	0.01	0.05	0.43	3.29	28.21
MVM	0.00	0.00	0.01	0.06	0.44	3.33	28.32

[8] S. Ramaswamy, E. W. Hodges, and P. Banerjee, *Compiling MATLAB Programs to SCALAPACK: Exploiting Task and Data Parallelism*, Proc. Int. Parallel Processing Symp. (IPPS-96), pp. 613–620, April 1996.

[9] M. Benincasa, R. Besler, D. Brassaw, and J. R. L. Kohler, *Rapid Development of Real-Time Systems Using RTExpress*, Proceedings of the 12th International Parallel Processing Symposium, pp. 594–599, March 30 - April 3, 1998.

[10] P. Drakenberg, P. Jacobson, and B. Kagstrom, *A CONLAB Compiler for Distributed Memory Multicomputer*,Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, vol. 2, pp. 814–821, 1993.

[11] A. E. Trefethen, V. S. Menon, C. C. Changm G. J. Caajkowski, L. N. Trefethen, *Multi-MATLAB: MATLAB on Multiple Processors,* Technical Report 96-239, Cornell Theory Center, Ithaca, NY, 1996.

[12] M. J. Quinn, A. Malishevsky, N. Seelam, Y. Zhao, *Preliminary Results from a Parallel MATLAB Compiler*, Proc. 12th International Parallel Processing Symposium, March 1998, pp. 81-87.

[13] M. J. Quinn, A. Malishevsky, N. Seelam, *Otter: Bridging the Gap between MATLAB and ScaLAPACK*, Proc. 7th IEEE International Symposium on High Performance Distributed Computing, August 1998.

[14] E. Rijpkema, E. Deprettere, B. Kienhuis, *Compilation from Matlab to Process Networks*, Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99), 1999.

[15] V. Menon and K. Pingali,*A Case for Source Level Transformations in MATLAB*, tech. rep., Department of Computer Science - Cornell University, 1999.

[16] J. Hollingsworth, K. Liu, P. Pauca, *Parallel Toolbox for MATLAB*, Technical Report, Wake Forest University (1996).

[17] S. Pawletta, W. Drewelow, P. Duenow, T. Pawletta, M. Suesse, *A MATLAB Toolbox for Distributed and Parallel Processing*, Proc. MATLAB Conference, Cambridge, MA, 1995.

[18] G. Almasi, C. Casaval, and D. A. Padua,*MATMarks: A Shared Memory Environment for MATLAB Programming*, tech. rep., University of Illinois - Computer Science Department, 1999.

[19] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, *Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems*, J. Parallel and Distributed Computing,Vol.59(2),Nov. 1999, pp. 107-131.

[20] M. A. Iverson, F. Ozguner, *Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment*, 7th IEEE Heterogeneous Computing Workshop (HCW'98), 1998, pp. 70-78.

[21] O. H. Ibarra, C. E. Kim, *Heuristic algorithms for scheduling independent tasks on nonidentical processors*, J. ACM, Vol.24(2), Apr. 1977, pp.280-289.

[22] C. Leangsuksun, J. Potter, S. Scott, *Dynamic task mapping algorithms for a distributed heterogeneous computing environment*, 4th IEEE Heterogeneous Computing Workshop (HCW'95), 1995. pp.30-34.

[23] M. Maheswaran, T. D. Braun, H. J. Siegel, Heterogeneous distributed computing, *in* "Encyclopedia of Electrical and Electronics Engineering",1999, Vol.8,pp.679-690.

[24] Condor Project Home Page, http://www.cs.wisc.edu/condor/

[25] Berkeley NOW Project Home Page, http://now.cs.berkeley.edu/

[26] Beowulf Project Home Page, http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html

[27] "MathWorks Home Page," tech. rep., MathWorks Inc., www.mathworks.com.

[28] "Mathtools Home Page," tech. rep., MathTools Corporation, www.mathtools.com.

[29] The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760–1500, USA, *Using MATLAB—The Language of Technical Computing*, Jan. 1997. Using MATLAB (version 5.0).