FACTA UNIVERSITATIS (NIŠ)
Ser. Math. Inform. 17 (2002), 75–106

AN ARCHITECTURE FOR PARALLEL INTERPRETATION OF ABSTRACT MACHINE LANGUAGES

Mark L. Manwaring and Veljko D. Malbaša

Abstract. An innovative central processor architecture is described that exploits the innate parallelism found in the machine language interpretation process. A new instruction encoding method, that facilitates the design of pipelines, is used to design the instruction sets of the controller, memory, and execution units. The problem of incorporating pipelined parallelism with other forms of implicit parallelism is discussed. Performance profiles of nine benchmark programs, obtained by using a cycle-level simulator, show the advantages of the described architecture for parallel over an equivalent architecture for serial interpretation. Performance measurements of representative benchmark programs show that a speed up of about two is achieved compared to the traditional sequential machine language interpretation on a single processor.

1. Introduction

Traditional computer architectures impose two serious problems: (i) inefficient support for a complex interpretation phase and (ii) classic interpretation techniques, based on a sequential fetch-execute model, are not very suitable for interpreting higher level, or abstract, machine languages, because of the phenomenon of "interpretive overhead" [2]. We propose a parallel model of interpretation to overcome the limitations of the traditional, inherently sequential, interpretation model. We also describe an architecture that supports parallel model of interpretation. This type of the architecture shares some commonality with a class of decoupled computer architectures, [10].

Received November 16, 1997.
2000 *Mathematics Subject Classification.* Primary 68M07.

Program execution is a process which takes the high level language program as input and produces the results implied by the semantics of the program. Program execution is usually divided into two phases. In the first, compilation, phase each statement in the high level language program is translated into a semantically equivalent sequence of code in the instruction set of the processor. In the next, interpretation, phase the processor executes the sequence of code generated by compilation to produce the final results.

Certain features of the present generation of programming languages make them more amenable to interpretation than compilation. For example, if program execution is dominated by the compilation phase, then most of the binding is done during the compile time. Although it is effective in traditional imperative languages like Pascal or Fortran, the binding during the compilation cannot be applied with the same degree of success to a large number of modern languages. Collection oriented ([13]) and object oriented languages are examples that fall in this category of languages.

A complex interpretation phase is involved in the interactive systems that are becoming an increasingly important asset of many modern computer applications. In an interpretive environment, every time a program element is reached, its binding to the environment is established, used and then destroyed. In an interactive system, where the programming language is provided with conversational mode and edit statements can be formulated within the source language, fixed bindings can be counterproductive because they impede fast and flexible changes of parts of the source program, and normally require a recompilation of the whole program. Thus efficient support of interactive environments requires an extensive interpretation phase.

Even though a complex interpretation phase is preferred in the applications which require extensive runtime support and resource management, traditional computer architectures are very inefficient in providing support for the complex interpretation phase. One reason for this is that because relatively low semantic contents of the instruction sets of contemporary RISC- and even CISC-type processors causes the bus bandwidth bottleneck in computer system, [11]. Cache technology is extensively used to mitigate the effect of high instruction traffic.

However, the increasing cache size or changing cache organization will not always provide an effective solution to this problem because: (i) it can not reduce the high instruction count, (ii) the push for higher performance by issuing more than one instruction per clock cycle exacerbates the bus bandwidth problem, and (iii) the increasing disparity between on chip and off-chip access time, resulting from the slower growth in DRAM compared

to processor performances, [12]. These three factors establish a cut-off point after which cache technology becomes ineffective in offsetting the disadvantages of larger code size in RISC systems.

In the proposed architecture the interpretation process is decomposed into concurrent processes which are executed in parallel on a number of specialized processors which are connected by fast queues. The speedup resulting from the parallel operation is designed to counterbalance the penalty of interpretive overhead. This is a form of user transparent implicit parallelism. High semantic contents of an high level instruction set leads to a proportionately smaller code size and therefore imposes less stringent requirements on the processor-memory traffic.

The performance of the described architecture can be enhanced by incorporating other forms of parallelism, such as pipelining. Furthermore, for various reasons, pipelined architecture for parallel interpretation is much easier to implement than, for example, pipelined superscalar architecture. In the paper we also deal with the design and performance of a pipelined version of an architecture for parallel interpretation.

A cycle level simulator for an example of the machine for parallel interpretation of abstract machine languages has been written. The simulator executes programs written in the machine code. Various architectural features, component speeds and queue lengths can be specified in a parameter file which is read by the simulator. The simulator generates various sets of data characterizing the execution process. These includes total parallel and serial execution time, execution time of each component, types and number of pipeline stalls, address traces et cetera. The estimate of serial time is based on the execution time of the same program on an equivalent serial machine. The benchmark programs that are run on the simulator are chosen to represent different types of loads. The measurements of performances are presented in comparison with some standard configurations.

The paper is organized in the following manner. The next section describes the architecture that provides the hardware support for parallel interpretation. Section 3 presents the design of a pipelined version of an experimental architecture that embodies the principles espoused in Section 2. Finally, selected performance profiles of both pipelined and non-pipelined versions of the the architecture for parallel interpretation are presented and analyzed.

2. An Architecture for Parallel Interpretation

2.1. Approach to parallel interpretation

To explain a motivation for parallel interpretation consider a possible sequence of instructions in pseudo-machine code that is equivalent to the assignment statement in high level language: $C := 3 + A + B$.

$$
\begin{array}{rcll}
R1 & \leftarrow & MEM(x) & \{\text{x is the address of A}\} \\
R2 & \leftarrow & MEM(y) & \{\text{y is the address of B}\} \\
R3 & \leftarrow & \text{ADD R1 R2} & \\
R4 & \leftarrow & \text{ADD R3 \#3} & \\
MEM(z) & \leftarrow & R4 & \{\text{z is the address of C}\}
\end{array}
$$

In the machine language all the mappings have already been resolved, and the very primitive nature of each of the operations precludes any concurrent processing of individual elements. However, if we were interpreting the assignments directly, we would have to execute a number of operations, like mapping the symbolic names A, B, and C to their locations, evaluating the expression, and storing the result. Note also that if we had two units, a name mapper and an execution unit, some of the operations could be done in parallel. For example, the mapping of C could be done in parallel with the evaluation of the expression. If we further assumed that our execution unit can perform two additions at the same time we could have further **overlapped** the operations by having the name mapper unit fetch B while the execution unit was involved with the first addition. Thus, by retaining the abstractions provided by the language at the machine level, it becomes possible to find concurrency in their execution.

The processor architecture that realizes parallel interpretation of high level machine language is called a *Minimally Synchronized Architecture*. In this section we present a structural model of an MSA, its operational principles, and the relationship between its performance characteristics and architectural parameters.

A minimally synchronized architecture (MSA) M is a two-tuple:

$$M = (IP, CC)$$

where IP is a finite non-empty set of *information processors* and CC is a finite non-empty set of *communication channels*, see Fig. 1. An information

processor IP is a two-tuple:

$$IP = (A, S)$$

where A is a finite non-empty set of *tasks* or *actions* and S is a dimensionless scalar quantity that characterizes the normalized speed of the given IP. A communication channel CC is a two-tuple:

$$CC = ((IP_i, IP_j), C)$$

The ordered pair (IP_i, IP_j) represents a one way communication channel between information processors IP_i and IP_j, and C is dimensionless scalar quantity that characterizes the capacity of the communication channel.

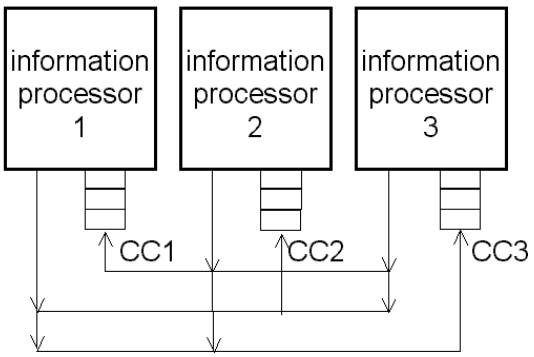

FIG. 1: MSA: an architecture for parallel interpretation

A task A is a 3-tuple:

$$A = (In, Out, T_{ex})$$

In and *Out* are finite non-empty subsets of IPs, and T_{ex} is the execution time of that task, measured in cycles of a global clock. If IP_i appears in the *In* set of a task A that is assigned to the IP_j then this implies that there should be a communication channel from IP_i to IP_j. Similarly, if IP_i appears in the *Out* list of a task B that is assigned to the IP_j then this entails a communication channel from IP_j to IP_i. The presence of an instruction processor in the *In* set of a task implies that the task must receive information from the IP before it can start activity. In the same vein, the IPs in the *Out* set of a task receive the results generated by the task. A task can not terminate if the transfers are not completed.

The MSA can be conceived as a collection of specialized processors which run concurrently and have well defined points of synchronization. The sequential process of interpretation is thus decomposed into a collection of concurrent cooperating processes. Each component of the system executes a pre-allocated component of the interpretation process. The process of the distribution of the tasks is done during the design phase of the MSA, and, once defined, becomes fixed. During its operation a particular machine might produce a data structure that is required by another machine, or it might need to consume data that is produced by another machine. The exchange of data takes place through buffered communication channels implemented as FIFO queues. The produce-consumer relation among different component IPs introduces synchronization points among the different units. For maximal concurrency the number of synchronization points must be minimal, hence the appellation "Minimally Synchronized Architecture."

2.2. The architecture of an example MSA

In this section we present an example MSA that illustrates the concepts developed. The MSA is designed to interpret a Pascal like language. The main reason for the choice is that Pascal and related languages have been extensively studied, and benchmark programs from these languages are widely available. These languages also provide a number of abstraction facilities: name space abstraction, the ability to specify an object by a symbolic name, data abstraction, the ability to define a relationship over a set of data, and procedural abstraction, the ability to abstract over any syntactic clause that can be written in-line in a program. In all compiler based implementations of such languages the mapping of the abstractions to the primitive machine language constructs is effected during compile time. However, in the proposed MSA these "high" semantic contents of these abstractions provide the opportunity for their parallel processing. Of course, in the framework of a predominantly static language, the rationality of resolving the mapping dynamically is questionable. The goal here, however, is to study some basic properties of MSA machines, and in this context the choice of the language is justified.

The proposed architecture consists of three IPs: the controller unit, the memory unit, and the execution unit, see Fig. 2. The controller IP communicates with the memory and execution IP via the CMQ (Control to Memory Queue) and CEO (Control to Execution Queue), respectively, and receives the results from the memory unit through the MCQ (Memory to Controller

Queue). At each cycle a 32-bit instruction parcel is brought in from the instruction cache to the instruction register (IR). If this is a leading parcel, the first byte of the instruction (the opcode) is copied over to the decode register. In case of a dependent parcel only the first four bits are copied on to the last four bits of the decode register. The content of the decode register is decoded and the *controller instruction* produces instructions for the memory and execution units.

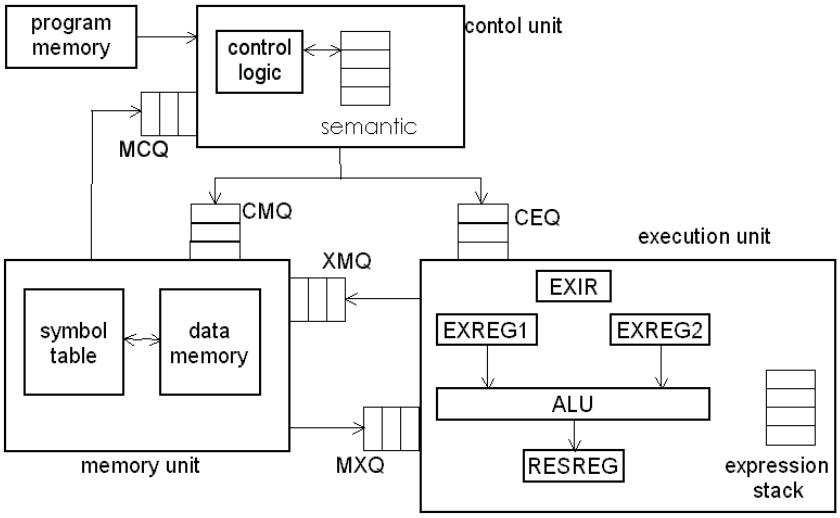

FIG. 2: Example of the MSA

A controller instruction can be decomposed into at maximum three memory and execution instructions. This consists of one address unit instruction and two execution unit instructions. These are written into the address instruction register ADREG and two execution registers EXREG1 and EXREG2, respectively.

The architectures of the controller, memory, and execution units, as well as the other details of the proposed architecture are given in [15].

3. Pipelined Architecture for Parallel Interpretation

3.1. PIPELINING

Pipelining increases computer performance by overlapping the execution of multiple instructions, [5]. This feat is accomplished by dicing each instruction into basic operations and dedicating individual processing units

(called *stages*) to each segment. The stages are connected with each other, usually via a staging register, and for a pipe. The number of stages present in a pipeline is called the depth of the pipeline. Assuming that the execution time in each pipeline stage is equal, the lower bound of the execution time of each instruction in the pipelined unit, called the *ideal execution time* is equal to the time per instruction on equivalent non-pipelined machine divided by the number of the pipe stages.

Hazards are pathological conditions in the operation of a pipeline where an instruction is prevented from executing at its designated clock cycle. Three kinds of hazards can be identified: *Data, Structural*, and *Control hazards.*

A **data hazard** occurs whenever some data object within the computer (e.g., register, memory locations, flag) is accessed or modified by two separate instructions that are close enough for their execution to be overlapped in the pipeline. Let the *domain* D_i of an instruction i be the set of all objects (registers, memory locations, flags, etc.) whose contents may affect the execution of the instruction i, and *range* R_i be the set of all objects whose contents may be modified by the execution of the instruction I. Three types of data hazards are distinguished between an instruction i and a successive instruction $i+1$.

- RAW (Read-After-Write) hazard is possible is $R_i \bigcap D_{i+1} \neq \varnothing$. RAW hazards occurs when the instruction $i+1$ attempts to read an object which is modified by the instruction i before the modification is complete.
- WAR (Write-After-Read) hazard is possible if $D_i \bigcap R_{i+1} \neq \varnothing$. WAR hazard is possible if instruction $i+1$ modifies some object before it is read by the instruction i.
- WAW (Write-After-Write) hazard is possible if $R_i \bigcap R_{i+1} \neq \varnothing$. This type of hazard occurs when both instructions i and $i+1$ attempt to modify the same object but the instruction i's modification occurs after that of the j's.

The simplest technique to resolve the data hazards is to "stall" the pipeline if a hazard is found, i.e. if the instruction $i+1$ is found to have a hazard possibility with a previously issued instruction i then the issuing of $i+1$, and all subsequent instructions is halted until such time when the hazard condition ceases to exist. A more ambitious solution is to stall the instruction $i+1$, but let subsequent instructions $i+2, i+3, \ldots$ proceed, if

they are free of potential hazards. A solution specific to RAW hazards, is to directly forward the data produced by the instruction i, which is required by the instruction $i+1$, to the instruction $i+1$ before the execution of the instruction i is complete. This is called *forwarding* or *short circuiting* and requires extra hardware for its implementation.

A **structural hazard** might rise in a situation where two or more instructions compete for the same hardware resources at a given clock cycle. For example, at a given clock cycle, the instruction i might want to write a result to the register bank, while the instruction $i+1$ might need to read the contents of a register. The usual solution is to let on of the competing instruction proceed while stalling the others. Depending on the frequency of a particular hazard, the designer might want to duplicate the resource of contention.

The **control hazards** happen in the presence of branch and jump instructions that disrupt the sequential flow of the instruction execution. The problem is that the target of a branch instruction become available only after the instruction is well into the pipeline. At this point several subsequent instructions have also been issued to the pipeline under the assumptions of linearity of execution. However, if the branch is now taken, this would require that the pipeline be "flushed" and the effects (if any) of the instructions already in the pipe, be undone, before the execution can be resumed from the new address. The simplest solution to this problem is to freeze the pipeline as soon as a branch or jump instruction is detected, which usually occurs during the decode stage. A more sophisticated technique involve predicting the outcome of the branch instruction and continuing the execution from a point based on the prediction. In case of a false prediction the pipeline has to be flushed.

A generic RISC pipeline consists of five stages: instruction fetch, instruction decode, execute, memory access, and write back, [3]. All types of the hazards are involved. A typical CISC type pipeline consists of six stages: instruction fetch, instruction decode, address generation, operand fetch, execution, and operand store or write back, [9]. The potential for hazards in this pipeline is enhanced because of the increased depth of the pipeline and the complexity of the instructions.

3.2. Pipelined MSA

Pipelined and superscalar architectures were both conceived to augment processor performance by exploiting implicit parallelism. the nature of the

parallelism that each exploits is, however, very different. Pipelining offers an economical way to realize the *temporal* parallelism, that is inherent in the process of instruction execution, by segmenting the process into consecutive subprocesses. Superscalar architectures, on the other hand, are said to exploit *spatial* parallelism, i.e., instruction level parallelism. These two orthogonal approaches can also be combined in the same processor to provide a higher degree of potential parallelism. An example which carries this to the extreme is the SIMP processor, [6] and [7], which consists of four identical instruction pipelines, each of which consists of five pipe stages thus enabling the processor, in theory, to attain an overall speedup of 20.

The parallelism exploited by an MSA is very different in flavor from that of temporal or spatial parallelism, and can be termed *structural parallelism*. It exploits the parallelism inherent in the structure of the interpretation process. The performance of an MSA architecture can also be enhanced by incorporating other forms of parallelism within its architecture: superscalar MSA or pipelined MSA. Furthermore, for various reasons, pipelined MSAs are much easier to implement than pipelined superscalar architecture. In a pipelined superscalar architecture, the detection of pipeline hazards are made more complicated, and their effect on processor performance is further exacerbated by multiple instruction issue and out of order execution. This is specially true for control hazards, which has been blamed for the poor performance of the SIMP processor.

The combination of structural and temporal parallelism in pipelined MSA is more benign because in machines that realize structural parallelism, the logical ordering of instructions is violated during execution. Structural and data hazards are not compounded by the presence of multiple pipelined units because (i) each pipeline is separate physical entity and do not share any common resources and(ii) the source and destinations of each pipeline are logically separated. Control hazards still impose a hefty penalty on the performance by introducing pipeline stalls, but this can be alleviated by a combination of software (delayed branches, software branch prediction etc.) and hardware (speculative execution, boosted execution etc) solutions.

Note that in all of the three pipelines the range and domain of the instructions do **not** overlap. The following table gives the domain and range objects for the three units. In the table IC stands for Instruction Cache, MCQ for Memory to Controller Queue, CMQ for Controller to Memory Queue, CXQ for Controller fo Execution unit Queue, DC for Data Cache, XMQ for Execution unit to Memory Queue, and MXQ for Memory to Execution unit Queue, see Fig. 3 and Fig. 4, [15].

pipeline	domain	range
controller	IC, MCQ	CMQ, CXQ
memory	DC, CMQ, XMQ	MXQ, MCQ, DC
execution	CXQ, MXQ	XMQ

The consequences of the logical and physical separation of the range and domain structures are profound, because it implies that there are **no** data hazards in these pipelines, and thus they do **not** suffer from the crippling effect of this type of hazards.

A quick inspection of the pipeline also reveals that these pipelines are devoid of structural hazards. The controller unit accesses the instruction cache only during one stage in the pipeline. Similarly, the memory unit has a single read/write access to the data cache at a given clock cycle. Memory accesses therefore can not cause structural hazards. At a given clock cycle the controller unit can perform at most three write operations, on to the memory queue and two writes to the execution queue. The memory queue is provided with one write and one read port, while the execution queue is provided with two write and one read port. This precludes any possible structural hazard resulting from access to the resources. Most multiple-pipelined superscalar architectures, in contrast, are characterized by an aggravation in the complexity of data and structural hazards when compared to their scalar counterparts.

Control hazards, however, are still present in this architecture, although their effect on performance is less pronounced when compared to superscalar architectures. The two controller instruction that generate this type of hazard in this architecture are the *GOTO* instruction, which denote an unconditional transfer of control, and the *LOOP* instruction wich is a conditional transfer instruction and whose outcome is predicated by the result of a previously issued relational instruction. This relational instruction is evaluated by the memory unit and its result is passed back to the controller unit via the memory-to-controller queue MCQ. The *GOTO* instruction can be detected in the ID stage of the pipeline and hence only the next instruction (which is in the IF stage) needs to be flushed from the pipeline. This introduces a pipe stall of one clock cycle. The effect of the *LOOP* instruction is more severe. The instruction can be detected as early as the ID stage, however the resolution of the branch is dependent on the evaluation of the relational instruction in the memory unit, and in the case when the relational instruction immediately precedes the branch instruction the evaluation can take from six to thousands of clock cycles (in case of a data cache read miss in the memory unit). These harsh effects can, however, be largely mitigated

by code optimization techniques like delayed branch and loop unrolling. In most cases the branch penalty can be reduced to 1 cycle. A further source of branch penalty are the $CALL$ and $RETN$ instructions. In each case the branch penalty is exactly one clock cycle.

The preceding discussion substantiates the claim that pipelined MSAs are significantly less prone to pipeline hazards than superscalar architectures.

3.3. Distributed encoding scheme and pipeline design

In relevant literature and research community the concept of CISC has been frequently equated to language-directed computer architecture and specially to any architecture that provides or is perceived to provide architectural support for high level language constructs. The unfortunate consequences of this misconceived identity is that there exists a tendency to brand any processor that provides silicon based support (as opposed to compiler based) for high level language constructs as "inefficient,' because of established inefficiencies of certain CISC processors.

We contend that increasing the abstraction level of an instruction set does not inexorably lead to a more complex instruction set. To substantiate this claim we will first analyze the nature of complexity of CISC type instructions. There are two major sources of complexities in CISC type instruction sets.

A major source of inefficiency in CISC style implementations is the absence of a clean separation of abstraction levels. In CISC type machine, both registers and memory locations are used as possible operand sources in the instruction set. A register set forms the fastest storage medium in the memory hierarchy, whose access time is comparable to the CPU cycle time. Memory based operands, on the other hand, require a much larger access latency. This disparity complicates the controller pipeline architecture since the machine has to contend with operands whose operational characteristics (access time, address encoding) are very diverse, [14]. The presence of such disparities in encoding and execution times among different instructions have serious repercussions on the design and performance of the instruction pipeline. Registers and memory locations which constitute different abstraction levels of the same entity (i.e. operand) are not conceptually and functionally separated at the processor level. This shortcoming will be referred to as the *abstraction complexity* of the instruction sets.

CISC type instructions also exhibit *structural complexity*. This refers to the high level of encoding of the instructions and their variable length. Highly encoded instructions take longer to decode which prolongs the criti-

cal delay path of the controller, thereby incriminating performance efficiency. Yet another facet of complex instructions is their varying length. The instruction set of a CISC machine contains instructions whose operation time can vary from a few cycles to hundreds of cycles. The impact of such multicycle operations on the performance of the pipeline can be very debilitating, [12].

Abstract instruction set of MSA avoids the pitfalls of CISC type instructions by (i) not overlapping the abstraction level of operands in the instructions, and (ii) using a Distributed Encoding Scheme (DES), [15], for instructions to solve the problem of structural complexity.

Each component of an MSA is characterized by its own abstract instruction set. In mapping a high level language object to a machine resource, it is usually necessary to map the object through a number of abstraction levels. for example, a high level language variable is usually mapped to a memory location, which can then be mapped to a register name. In an MSA each component IP would handle operands only at a given level of abstraction. For example, name abstraction could be handled on three different levels: the first deals exclusively with variable names, the second understands the concept of memory locations only, while the third is restricted to deal with register level only. The mapping of the variable name to the register location is effected by cooperative and concurrent operation of the three machines. Each IP manipulates only one level or aspect of abstraction. The abstraction complexity of CISC type instructions is thus avoided.

Another important aspect of pipeline design is the effect of the structural complexity of an instruction set on pipeline performance. An instruction encoding scheme, called *Distributed Encoding Scheme (DES)*, that strives to minimize structural complexity of MSA instruction set while retaining the compactness of a complex instruction set is presented in this section. The DES is derived from the observation that most complex instructions consist of a number of operations which are executed in a sequential fashion and which usually over-utilizes the resources of the pipeline. The central idea in DES is to decompose a complex instruction into several constituent parts, each of which encodes several operations from the operation set of the complete instruction, and is complex enough to fully utilize the resources of the pipeline. This structural unit of a complex instruction is called an *instruction parcel*. So, an instruction parcel is a structural quanta of complex instruction that can be fetched and issued as an independent instruction and whose semantic content is sufficient to properly utilize the resources of the pipeline. In DES, unlike conventional instruction formats where the opcode

that identifies the functionality of the instruction is usually confined to the leading bytes (one or two) of the encoding, the information is distributed among the constituent instruction parcels.

The instruction parcels can be differentiated into two categories, which are called *leading* and *dependent* parcels. The leading instruction parcel is the first instruction parcel in a multi-parcel instruction. Subsequent parcels in the instruction fall in the category of dependent parcels. A single bit in the encoding differentiates between the two types. The encoding of a leading instruction parcel consists of the two fields, the *major opcode* field that identifies the instruction, and when decoded, provides the relevant semantic information that determines the operations to be performed on the operands specified in the second *operand* field. A dependent instruction parcel also consists of two fields. The leading field is called the *minor opcode* field, which is usually smaller than the major opcode field, and when combined with a specified segment of the corresponding major opcode field, provides the information required to process the operands that are provided in the operand fields that follow. The leading and dependent instructions have a fixed field encoding which facilitates their decoding.

DES offers a good combination of compactness of code and ease of decoding. Decoding complexity is alleviated by the fixed field encoding of the instruction parcels. A reasonable compactness is also maintained by making the minor opcode field considerably smaller than the major opcode field. Pipeline efficiency is not, however, compromised because the instructions are processed parcel by parcel each of which is designed for efficient use of pipeline resources.

The DES concept is illustrated by an example design of the instruction PEXPR, that specifies a polish expression, and that is the most complex instruction in the MSA instruction set. The expression can be arbitrary long, and can consist of an arbitrary combination of operands and operators in arbitrary order. It is assumed that 4 bits are required to specify an operator and that 24 bits are required to specify a literal values or the address of an operand. The major opcode is specified by 8 bits, the last four bits of which is combined with the 4 bit minor opcode to yield the working opcode for dependent parcels. The instruction parcel is 32 bit long, which is the usual length for modern RISC type instructions. The instruction parcels for PEXPR that can be formulated under these constraints are given in the following table, where for each instruction the mnemonic name and the name and type of operand fields are given.

opcode	field 1	field 2	description
PSWR	VAR		leading parcel, variable operand
PSVI	VAR		leading parcel, indirect var. operand
PSVL	VAL		leading parcel, literal operand
PAVR	BASE		leading parcel, structured variable
POVL	OPER	VAL	dependent parcel, operator and literal
PVLO	VAL	OPER	dependent parcel, literal and operator
POVR	OPER	VAR	dependent parcel, operator and operand
PVRO	VAR	OPER	dependent parcel, variable and operator
POVI	OPER	VAR	dep. parcel, operator and indirect operand
PVAL	VAL		dependent parcel, literal operand
PVAR	VAR		dependent parcel, variable operand
PVRI	VAR		dependent parcel, indirect variable
POPP	OPER	OPER	dependent parcel, two operators
PAOD	VAL		dep. parcel, literal offset for string operand
PAOV	VAR		dep. parcel, variable offset for string operand

Note that the first four are leading instruction parcels while the rest belongs to the dependent category. A polish expression will be encoded as a combination of these parcels with the proviso that the encoding must start with on of the four possible leading parcels. Note that while a certain amount of independence can be ascribed to a parcel, it is not an independent instruction because its import is only valid in the context of the complete instruction of which it is a part.

As an example, the encoding of the representative polish expression:

$$(A + B) \star C \star D \star E$$

in the compact PEXPR form is given as PEXPR $AB + C \star DE \star\star$ and in the DES format as:

PSVR	A	
PVRO	B	+
PVRO	C	+
PVAR	D	
PVAR	E	
POPP	+	+

If the size of the variables is 24 bits, and size of the operators is 4 bits, then the code size for the compact PEXPR form is $24 \cdot 5 + 4 \cdot 4 = 144$ bits, and the corresponding size for DES form requires 192 bits, that is about 32%

more than the compact form. This overhead in code size is quite acceptable when compared to the more than twofold increase in RISC code size over that of CISC code size. More importantly, the performance of the pipeline has not been compromised because each instruction parcel exhibits the same structural simplicity that distinguishes RISC type instructions.

A detailed description of the pipelined MSA controller instructions is given in [15].

3.4. Controller unit

The architecture details of the controller unit of the proposed pipelined MSA is presented in the Fig. 3, and the instruction set is detailed in the reference [15].

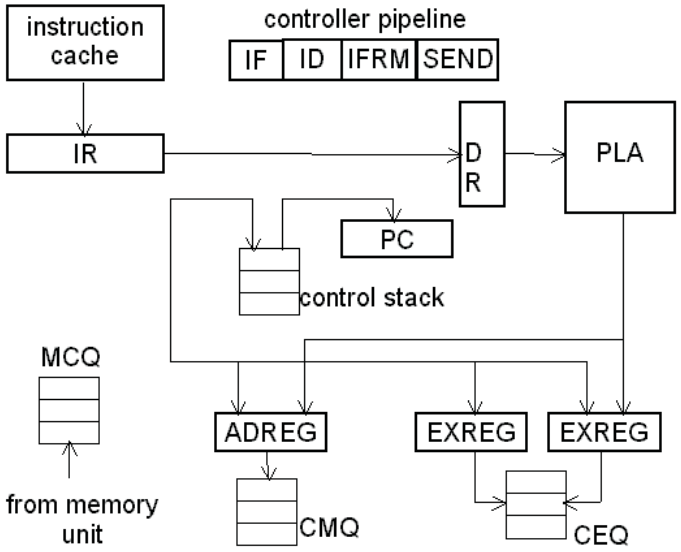

Fig. 3: The pipelined controller unit of the MSA

It is important to note that all of the three processors, the controller, the memory, and the execution unit, are pipelined. The structure of the pipelines have been discussed in the previous section. In this and following sections the overall architecture of the units will be discussed, which also includes the instruction set of the units. The instruction set architecture of each unit was designed in accordance with the DES concept.

The controller communicates with the memory and execution processor via the CMQ and CEQ queues, respectively, and receives the results of relational operations from the memory unit through the MCQ. At each cycle a 32-bit instruction parcel is brought in from the instruction cache to the instruction register (IR). If this is a leading parcels, the first byte of the instruction, i.e. the opcode, is copied over to the decode register. In the case of a dependent parcel, only the first four bits are copied on to the last four bit of the DR. The contents of the DR is decoded by the PLA. When decoded, a controller instruction produces instructions for the memory and the execution units. The maximum number of memory and execution instructions that a controller instruction can be decomposed into is three, which includes on address unit instruction and two execution unit instructions. These are written into the address instruction register (ADREG) and the two execution instruction registers (EXREG1 and EXREG2), respectively.

The instructions are formed in the following fashion. The output of the PLA consists of up to a maximum of three fields, each of which is copied on to one of the three registers. The operand field of these registers are copied from the operand field of the IR. When a controller instruction is in the write stage the contents of the non-empty ADREG, EXREG1 and EXREG2 are copied to the CMQ and CEQ.

3.5. Memory unit

The architecture of the memory unit is presented in the Fig 4, and some of its instructions are described in Table 3.1.

The memory unit receives its instruction from the controller unit trough the CMQ. At each clock cycle an instruction is fetched from the CMQ and copied to the memory instruction register (MIR). If the queue is empty, an NOP instruction is generated. The address part of the instruction is copied onto the memory address register (MAR). If the instruction requires a memory read, then the data is read from the data cache (DC) to the load memory data register (LMDR), and is subsequently written into MXQ. If memory write is indicated, the contents of store memory data register (SMDR) is written to the DC. The value in the SMDR has been copied from the MIR during an earlier cycle (ID).

In both cases, the address of the load/store operation is fetched from the MAR. The second category of instructions that the memory unit handles is the comparison instructions. In this case the instructions are fetched from the CMQ, the relation evaluated by the memory unit comparator, and the

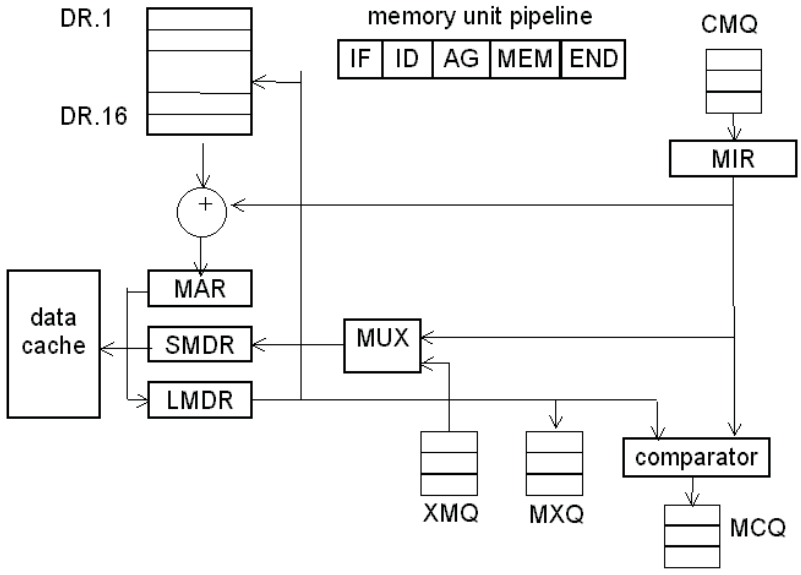

FIG. 4: The pipelined memory unit of the MSA

result is written back to the controller via the MCQ. The third category of instructions that is handled by this unit are the subroutine call and return. The processing of these instructions involves the creation or deletion of data stack frames and will be explained later.

3.6. EXECUTION UNIT

The execution unit fetches its instruction from the CEQ. As in the case of the memory unit, an empty CEQ generates a NOP. The instruction set of the execution unit is shown in the following table.

instruction		action implied
IVAL	VAL	push value operand on the expression stack
QVAL		push a pointer to QITEM on the expression stack
OPER	<OPERATOR>	execute the operation specified by the OPERATOR field on the first two items on the expression stack
SEND		send the value in RESREG to the EMQ

Table 3.1: Description of instructions

memory instruction			action implied	pipe delay
ACMVR0	DR.M	OFFSET	compare variable with 0	0 cycle
ACMVIO	DR.M	OFFSET	compare indirect variable with 0	1 cycle
ACMVR1	DR.M	OFFSET	compare variable with 1	0 cycle
ACMVI1	DR.M	OFFSET	compare indirect variable with 1	1 cycle
ARLS	DR.M	OFFSET	relation operation of type *var rel var*	0 cycle
RLGT	VALORVAR		> comparison and second operand	0 cycle
RLGE	VALORVAR		>= comparison and second operand	0 cycle
SNVR	DR.M	OFFSET	send variable to MEQ	0 cycle
SNVI	DR.M	OFFSET	send indirect variable to MEQ	1 cycle
STQV	DR.M	OFFSET	store variable from EMQ in address	0 cycle
STQI	DR.M	OFFSET	store variable from EMQ in address ind.	1 cycle
STIV	DR.M	OFFSET	store immediate data in address	0 cycle
STII	DR.M	OFFSET	store immediate data in address indirect	0 cycle
STID	VALUE		literal value from the previous instruction	0 cycle
SNAR	DR.M	OFFSET	send structured var to MEQ, 1st parcel	0 cycle
SNOD	OFFSET		2nd parcel, offset is literal	0 cycle
SNOV	DR.M	OFFSET	2nd parcel, offset is variable	0 cycle
STAR	DR.M	OFFSET	store structured var from EMQ, 1st parcel	0 cycle
STOD	OFFSET		2nd parcel offset is literal	0 cycle
STOV	DR.M	OFFSET	2nd parcel offset is variable	0 cycle

3.7. Address encoding

MSA supports two types of memory objects: (i) simple scalar objects and (ii) structured objects. All simple variable operands are encoded by a pair of values which are combined together to form the address of the variable, i.e. to bind the variable to a memory location. The 24 bit address consists of a four bit *display* field and a 20 bit *offset* field. The display field determines the environment in which the variable is defined. The display field is decoded to point to a display register in the memory unit which contains the base address of this environment. The actual address is calculated by adding the 20 bit offset to the base address specified by the relevant display register. Since the offset field is 20 bit long this implies that at most 2^{20} local variables are allowed in each environment. A 4 bit value for the display register implies that a maximum *static* nesting of 16 levels are supported by the MSA.

Structured variables are characterized by a base address, display and dimension. Each component of a structured object is further specified by an offset. The address of a component of a structured object is formed by adding three components: (i) base address of the data frame which contains the object and which is determined by the display register, (ii) base address of the object, and (iii) offset of the component. In accordance to the DES principle, the base address and the display level of the structured object is specified by a leading parcel, and the offset is specified by a dependent parcel.

3.8. Procedure abstraction

The instruction set of MSA has been augmented to support the procedure abstraction. A distributed version of the Johnston's Contour model has been adopted for this purpose, [1], [4]. The management of the run-time environment necessary for this feature is distributed between the controller and memory units. The conventional activation stack is split into two stacks in the MSA. The control stack, which stores the return addresses, is maintained by the controller unit, and the *data stack*, which provides the space for the local variables is managed by the memory unit. Access to non-local variables is provided through the display mechanism. The displays are maintained in 16 display registers (DR.1 - DR.16) in the memory unit. Apart form the display registers, the memory unit also contains a HIGHMEM register, which points to the top of the data stack, and a CURDISP register which points to the display register associated with the current environment.

The layout of the activation record of a procedure in the data stack is organized in the following way. The first entry contains the old value of the display register m which was saved at this location when the frame was created. The rest of the frame consists of parameters passed to the procedure and local variables, which includes both scalar and array type variables. Heap type storage is not required because dynamic variables are not supported in the MSA.

3.9. Control transfer instructions

Both unconditional and conditional control transfer instructions are present in the instruction repertoire of the MSA. GOTO implements control transfer. The conditional control transfer statement is formed of two sets of instructions, the LOOP instruction and the relational instructions transfer statement. First the condition on which the transfer is predicated must be evaluated. Once the value is known, actions pertaining the execution of control transfer can be undertaken, if required. The relational instructions are responsible for the evaluation of the branch condition. This set of instructions is DES encoded to alleviate structural complexity. The two leading parcels are RLS1 (for relation of the type VAR REL VAL) and RLS2 (for the relations of the type VAR REL VAR). The rest of the instructions specify the kind of relational operation and the second operand of the expression. the interpretation of the second operation (variable or literal value) is contingent upon the preceding parcel. Also included in the category of relational instruction are four single word instructions which compare the immediate operand for equality with 0 or 1.

4. Performance Characteristics

The goal of this part of the project was to come up with some performance measures and to study the effects of the variation of different and significant architectural parameters on both the pipelined and the non-pipelined MSA machines. The measures are not presented in absolute metrics, because these measurements are highly dependent on technological factors and hence are more applicable to either finished products or those at the last stages of development. The goal is to study the effects of the variation of different and significant architectural parameters on machine performance. Machine performance is measured in units of speedup over an equivalent serial machine and the proposed machine.

4.1. Performance indicators

The measurements of performance are presented in comparison with some standard configurations. An *equivalent serial machine* of a pipelined MSA is a serial machine whose instruction set is identical to the instruction set of the controller unit of the MSA, and for which the execution time of each instruction is defined by the sum of the execution times of the semantic actions activated by the controller instruction in the MSA. Therefore, the equivalent serial machine does not incur the penalty of synchronization of an MSA. A *base MSA* is a machine, all of whose component IPs have a speed of unity and all the queues of which are of length one. A metric that measures the effectiveness of the MSA configuration is the ratio of execution time of a serial equivalent machine to that of a base MSA.

A *component speedup* is the speedup factor of a given component. This is an architectural attribute specified as a design parameter. The component speedup characterizes the computational capability of the component which can be augmented by different mechanisms, e.g. by investing more silicon in the unit (pipelined unit, faster circuits etc.) or by using a different technology (ECL, GaAs). The parameter component speedup thus denotes a quantitative characterization of the increase in the computational capabilities.

A *System speedup* for an MSA of a given configuration is the ratio of the execution time of a program on a base machine to the execution time of the same program on an MSA of the given configuration.

4.2. Benchmark programs

The workload of the simulation of the non-pipelined MSA consists of two programs. The first program is a differential equation program, and involves a large number of floating point operations. This is a fourth order Runge Kutta solution to the equation $dy/dx = y + xy$. The solution was calculated for $x = x_0$ to $x = x_{final}$ with a step size of h. These the parameters were given as inputs to the program. The second program calculates the number of prime numbers between 1 and 100 five times using the Eratosthenes Sieve algorithm. This program is dominated by control flow statements with minimum computations. The programs have been chosen to create an asymmetric load for the MSA in the sense that for each program either the execution unit (program #1) or the memory unit (program#2) is predominantly used. Therefore it conjectured that this program driven

minimal concurrency would provide estimates on the lower bound of the performance of this class of machines. For each program several architectural attributes (component speedup, queue length, for example) were varied to measure their impact on the machine performance.

The workload of the simulation of the pipelined MSA consists of seven programs chosen to represent different types of workloads. The first five Livermore loops represent a load characterized by intensive floating point processing, and results in a high degree utilization of the execution unit, and also provide a moderate level of loading of the memory units. Loops 1 and 2 involve evaluation of long expressions which augments the level of parallel execution of the memory and execution units through the mechanism of use order renaming. The next benchmark program involves the evaluation of the Ackerman's function. It is a memory intensive program, characterized by deep recursive calls and in which the execution unit is almost never used and results in a completely unbalanced loading of the components. The last program is a list insertion routine which inserts several data items in a doubly linked list. This program is characterized by shallow calling depths and heavy utilization of the memory unit at the expense of the execution unit, although not to the extent of the previous program.

4.3. Simulator

To provide a testbed for exploring the performance of the proposed architecture a cycle level simulator of the described pipelined MSA has been implemented, [15]. The overall structure of the simulator is given in Fig. 5. The simulator reads a hardware specification file which defines various architectural attributes of the machine to be simulated. Parameters that can be specified include the various queue lengths and relative speeds of the different units. The program consists of four main modules, three of which simulate the controller, memory and execution units, and the fourth, the scheduler, coordinates the activities of the different units and also collects the data at every clock cycle. Several utility programs are also provided. A compiler translates the high level language programs into controller machine language programs. Several result analyzers process the large data files generated by the simulator and compute the performance parameters.

The simulator executes programs written in the machine code. The simulator generates various sets of data characterizing the execution process. These includes total parallel and serial execution times, execution time of each component, types and number of pipeline stalls, address traces, etc.

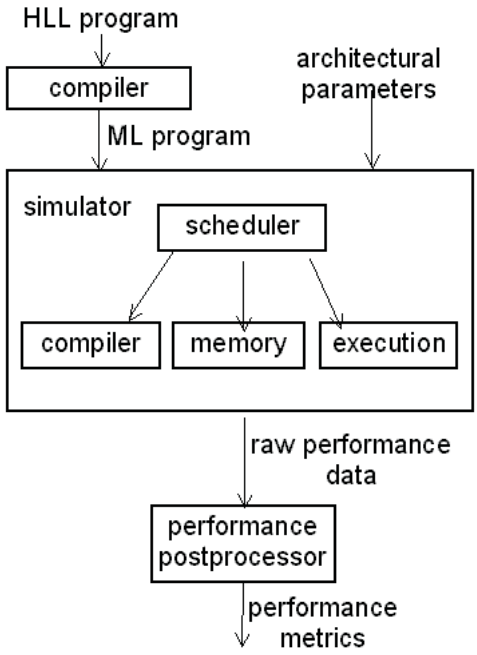

FIG. 5: The structure of the simulator

The estimate of serial time is based on the execution time of the same program on an equivalent serial machine.

4.4. Results of performance measurements

System speedup: pipelined MSA. The speedup of the base machine over the equivalent serial machine for different benchmark programs is given in the following table.

benchmark:	Livermore loops					Ackerman	List insertion
	#1	#2	#3	#4	#5		
speedup	1.72	2.37	2.83	2.93	2.5	2.56	2.1

Variation of the queue length: pipelined MSA. For this set of experiments the length of the different queues were varied over a range of 1 to 10 with increments of 1. The effects of the instruction queues (CMQ and CEQ) were studied with unbounded MXQ and XMQ, and vice versa,

the effects of the MXQ and XMQ were studied with unbounded CMQ and CEQ. This was done in order to decouple the effects of the various queues on performance. The results obtained, with minor variations, were almost identical for all the programs in the benchmark suite.

Varying the length of the instruction queues from the minimum (1) to the maximum (10) value results in an improvement of the performance by a factor of 1.05 over the base machine. Moreover, this value was reached by a queue size of 3, after which increasing the queue size had no effect on the performance. This observation was mirrored in the case of study involving the other set of queues. The maximum speedup obtained was 1.2, and the saturation point was reached by a queue length of 3.

The reason for this minimal increase is easily explained. The controller unit for each instruction it executes produces data items (instructions) that must be processed by the execution and memory units. Therefore, a close lock-step type synchronization is established between the controller and the other two units which minimizes the effect of the capacity of the communication channels.

Variation of the component speedup: pipelined MSA. This set of experiments shows the effect of varying the speed of the memory and execution units over a range of 1 to 16 (with the controller speed as the base speed) on the program execution time.

Different combinations of memory and execution unit speed are considered. An example of a typical diagram obtained running this set of experiment is given in Fig 6 that shows the system speedup versus the memory unit speedup. The Livermore loop programs are characterized by intensive floating point computations and as such the greatest benefit is reaped by enhancing the processing power of the execution unit. This observation is completely corroborated by the system speedup curves. Compared to the gain obtained by increasing the speed of the execution unit, the net gain in processing time obtained by increasing the processing power of the memory unit is relatively small.

The Ackerman benchmark and the List insertion benchmark provide a different workload and therefore radically different performance profiles. The base speedup in the case of the Ackerman function is 2.1 and for list is 1.9, considerably larger that the first Livermore loops (1.73). In view of the fact that both this benchmark create a highly unbalanced load this result might seem surprising. The reason for this apparent discrepancy can be explained by the fact that the exclusion of the execution unit from the processing also excludes all the various stalls associated with this unit and results in a con-

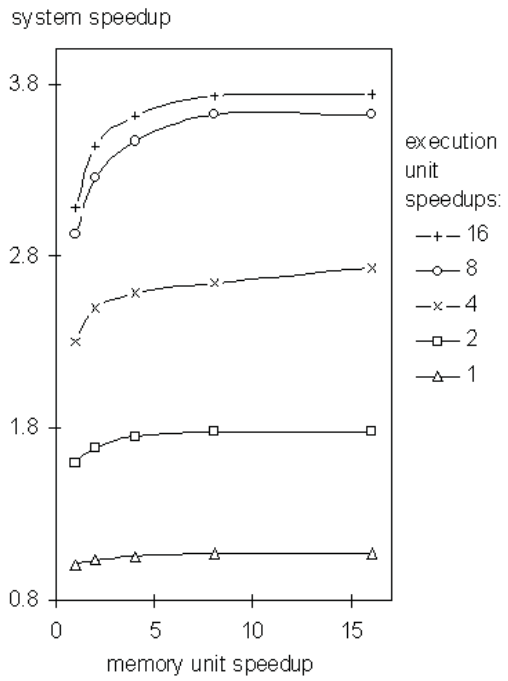

FIG. 6: System speedup vs memory unit speedup (Livermore #1)

figuration where the controller and memory units operate very much like a two stage pipeline which results in the doubling of the combined throughput of the system. The same observation applies to the case of the List insertion program, albeit to a lesser extent.

System speedup: non-pipelined MSA. The value for the concurrency index obtained for the differential equation program (program #1) is 1.9 and for the program that calculates the prime numbers (program #2) is 1.5, as shown in the following table.

benchmark program	program #1	program #2
concurrency index	1.9	1.5

The base concurrency that can be exploited in these programs is limited by the unbalance load. In program #1 the detrimental The base concurrency that can be exploited in these programs is limited by the unbalanced load. In program #1 the detrimental effect of this asymmetry is mitigated by the parallel execution of the memory and the execution units in the evaluation

of polish expressions. In the program #2 the more stringent synchronization barriers reduce the concurrency index to 1.5.

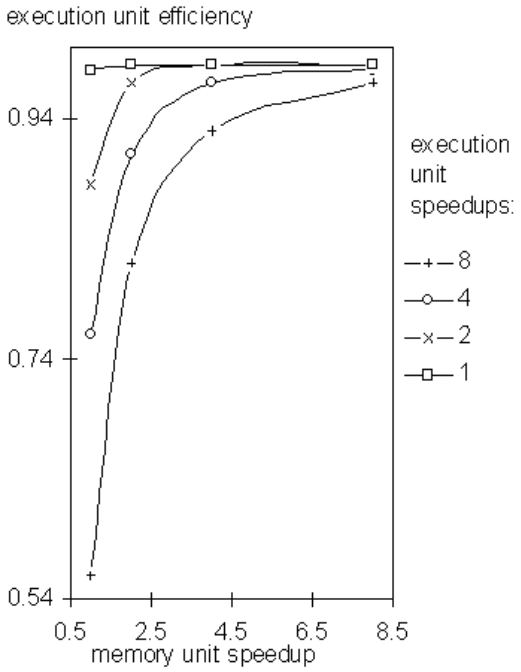

FIG. 7: Execution unit efficiency vs memory unit speedup (program #1)

Variation of component speedup: non-pipelined MSA. The next set of results demonstrates the effect of component speedup on the system speedup and component efficiency. Fig. 7 and Fig. 8, obtained for program #1, show that for a given value of speedup of the execution unit, increasing the speed of the memory unit improves the efficiency of the execution unit and reduces the efficiency of the memory unit. This result follows from the observation that since the memory unit is, for a large fraction of the program execution time, waiting on the execution unit, therefore any further increase of the memory unit speedup factor would add on to the number of memory unit waiting cycles, which translates into reduced efficiency. The converse is the case for the execution unit: increasing the speedup factor of the execution unit decreases the number of waiting cycles for the memory unit and boosts its component efficiency index. Since the program #1 involves a significant amount of floating point computation, the execution unit forms the bottleneck of the system as both the memory and the controller units

are perennially blocked waiting for the results of this unit.

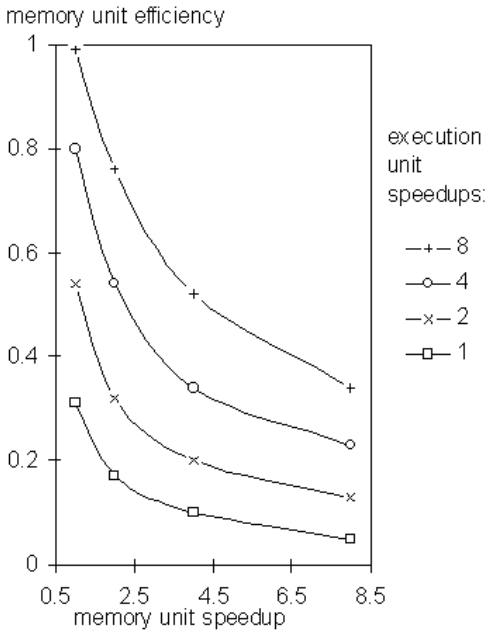

FIG. 8: Memory unit efficiency vs memory unit speedup (program #1)

The deleterious effects of the execution unit bottleneck can be compensated for by increasing its component speed as can be seen in Fig. 9. However, for a given execution unit speedup, increasing the memory unit speed recreates a dynamic load imbalance between the two units which is indicated by the levelling off of the system speedup curves for higher memory speedup.

Similar results are obtained for benchmark program #2. In this case, because, of the preponderance of control transfer instructions, the memory unit constitutes the bottleneck. The other components spend a major fraction of the program execution time in the blocked state. It, therefore, follows that augmenting the execution unit speedup has minuscule effect on the system speedup. By the same token, increasing the memory unit speedup factor has positive impact on the system speedup. However, the speedup factor is limited to a maximum value of 2.2 because of the fact that the condition evaluation operation is a simple operation (in terms of execution time) and the number of cycles that can be saved by speeding up this operation is relatively small. A very marginal increase in controller efficiency is observed when varying the speedup factor of the components, and can be attributed

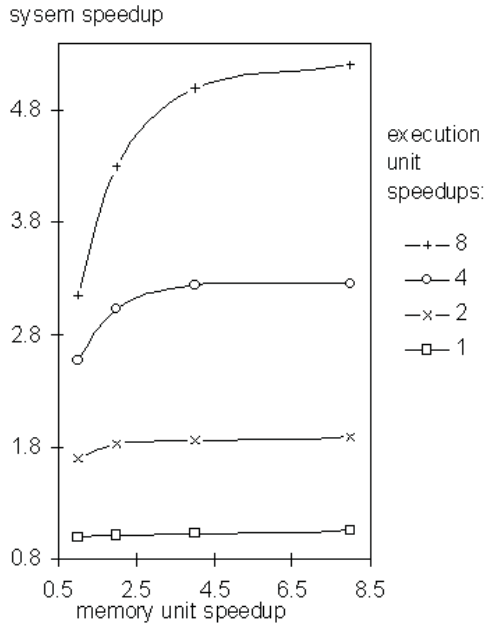

FIG. 9: System speedup vs component speedup (program #1)

to the fact that the execution unit plays an insignificant role in the overall computation.

Variation of the queue length: non-pipelined MSA. A set of simulations was performed to characterize the dependency of performance on queue length for the benchmark programs. The results are presented in Table 4.1 and Table 4.2. For program #1, changing the the queues length results in higher performance. The savings emanate from reducing the blocked time of the memory unit, which can slip ahead of the execution unit because of a more "spacious" buffer. For program #2, increasing the queue lengths fails to demonstrate any tangible performance advantage, probably because of the stringent synchronization requirements of the control dominated program.

5. Conclusion

We have discussed a new class of architecture, called *Minimally Synchronized Architecture* or MSA, that provides efficient hardware support for

Table 4.1: Effect of MXQ/XMQ queue length on performance (program #1) (Parameters: CMQ = CEQ length = 40; memory unit speed = 8; execution unit speed = 8)

MEQ/EQM length	1	2	3	4	5	8
Performance (clock cycles)	211425	210086	210886	210886	210886	210886

Table 4.2: Effect of CMQ/CEQ queue length on performance (program #1) (Parameters: CMQ = CEQ length = 40; memory unit speed = 8; execution unit speed = 8)

CMQ/CEM length	1	2	3	4	5	8
Performance (clock cycles)	220740	212228	210887	210886	210886	210886

high level machine language interpretation, achieved by using structural parallelism. The structural parallelism exploits the parallelism inherent in the structure of the interpretation process. The performance of an MSA can also be enhanced by incorporating other forms of parallelism within its architecture: superscalar MSA or pipelined MSA. Furthermore, for various reasons, pipelined MSAs are much easier to implement than pipelined superscalar architecture.

The paper presents the design details of a pipelined MSA with three pipelined units: controller, memory and execution units. Note that in all of the three pipelines the range and domain of the instructions do **not** overlap. The consequences of the logical and physical separation of the range and domain structures are profound, because it implies that there are **no** data hazards in these pipelines, and thus they do **not** suffer from the crippling effect of this type of hazards.

The pipelined units were designed by using the Distributed Encoding Scheme (DES) that was derived from the observation that the most complex instructions consists of a number of operations which are executed in a sequential fashion and which usually over-utilize the resources of the pipeline. The DES retains structural simplicity of the instruction set at the cost of an acceptable increase in code size.

Three different groups of experiments were organized by using the cycle level simulator. First group of the experiments demonstrate the effect of component speedup on system speedup and component efficiency. For a given value of speedup of the execution unit, increasing the speed of the memory unit improves the efficiency of the execution unit and reduces the efficiency of the memory unit. The converse is the case for the execution unit: increasing the speedup factor of the execution unit decreases the number of waiting cycles for the memory unit and boosts it component efficiency index.

A second set of simulations was performed to characterize the dependency of performance on queue lengths. It was shown that expanding the queues results in higher performance. For the program that calculates the number of prime numbers between 1 and 100, increasing the sizes of the queues however has minimal effect, because of the stringent synchronization requirements of the control dominated program.

Finally, the third set of simulations shows the speedup relative to the equivalent serial interpretation for five different benchmark programs. The speedup in case of Ackerman function is 2.56, for the List insertion is 2.1, and for the Livermore loops is about 2.5.

Performance measures show that the system speedup of about two is obtained relative to the equivalent serial interpretation for five different benchmark programs. Analysis of the performance curves also reveals information about the optimal speed configurations. For example, for the Livermore loops, it can be seen that 90% of the maximal performance can be obtained by controller/memory/execution unit speed ratio of 1/14/8. Further increase in the component speed does not result in a proportionate increase in system performance. Of course, the optimal speed configuration is dependent on the load characteristics. In the case of the Ackerman function, the optimal speed ratio would be 1/4/1/ which results in 95% of the maximal attainable speed.

REFERENCES

1. A. Aho, R. Sethi, and J. Ullman: *Compiler Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. D. Eddy and J. Campenhout: *Interpretation and Instruction Path Coprocessing*. MIT Press, 1990.
3. J. Henessy and D. Patterson: *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 1990.
4. J. Johnston: *The contour model of block structured processes*. SIGPLAN Notices **6** (1971), 121–145.

5. P. Kogge: *The Architecture of Pipelined Processors*. McGraw-Hill, 1981.
6. M. Kuga, M. Kazuaki, and S. Tomita: *DSNS: Yet another superscalar processor architecture*. Computer Architecture News **19**, No. 4 (1991), 14–29.
7. M. Kazuaki, N. Irie, M. Kuga, and S. Tomita: *SIMP: A novel high-speed single-processor architecture*. Proc. 16th Annual Int'l Symposium on Computer Architecture, June 1989, pp. 78–83.
8. M.L. Manwaring, M.F. Chowdhury, and V.D. Malbaša: *An Architecture for Parallel Interpretation: Performance Measurement*. Euromicro '94, Liverpool, England.
9. D. Phillips: *The Z8000 Microprocessors*. IEEE Micro, December 1985, pp. 23–36.
10. J. Smith: *Decoupled access/execute computer architecture*. ACM Trans. Computer Systems **2**, No. 4 (1984), 289–308.
11. J. Davidson and R. Vaughn: *The effect of instruction set complexity on program size and memory performance*. Proc. ASPLOS, 1987, pp. 60-64.
12. J. Hennessy and D. Patterson: *Computer Architecture*. Morgan Kaufman, 1990.
13. G. Sabot: *The Parlation Model*. MIT Press, 1988.
14. D. Alpert, A. Averbuch, and A. Danieli: *Performance comparison of load/store architectures*. 17th Intl. Symposium on Computer Architecture, Seattle, 1990, pp. 172–181.
15. M. F. Chowdhury: *Parallel Interpretation of Abstract Machine Language*. Ph.D. Thesis, Washington State University, Pullman, WA, 1993.

School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752, USA

School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752, USA
and
Department of Electrical Engineering
University of Novi Sad
21000 Novi Sad, Yugoslavia

e-mail: malbasa@uns.ns.ac.yu