Exploiting Fine Grain Parallelism in Prolog

Ashok Singhal

ABSTRACT

The potential usefulness of Prolog has motivated attempts to achieve higher performance than the fastest sequential Prolog systems currently available. Several researchers have attempted to do this by exploiting parallelism. The fundamental tradeoff is exposing more parallelism versus reducing the overhead incurred by parallel execution. The high execution overhead of the complex memory management and task management schemes used in current parallel Prolog systems leads to the following questions: (1) What forms of parallelism can be effectively exploited with memory management and task management schemes that have low overhead? (2) How much speedup can be obtained by exploiting these forms of parallelism? (3) What architectural support is required to exploit these forms of parallelism?

The goals of this research are to provide answers to these questions, to design a Prolog system that automatically exploits parallelism in Prolog with low overhead memory management and task management schemes, and to demonstrate by means of detailed simulations that such a Prolog system can indeed achieve a significant speedup over the fastest sequential Prolog systems.

We achieve these goals by first identifying the large sources of overhead in parallel Prolog execution: side-effects caused by parallel tasks, choicepoints created by parallel tasks, task creation, task scheduling, task suspension and context switching.

We then identify a form of parallelism, called flow parallelism, that can be exploited with low overhead because parallel execution is restricted to goals that do not cause side-effects and do not create choicepoints. We develop a master-slave model of parallel execution that eliminates task suspension and context switching. The model uses program partitioning and task scheduling techniques that do not require task suspension and context switching to prevent deadlock. We identify architectural techniques to support the parallel execution model and develop the Flow Parallel Prolog Machine (FPPM) architecture and implementation.

Finally, we evaluate the performance of FPPM and investigate the design tradeoffs using measurements on a detailed, register-transfer level simulator. FPPM achieves an average speedup of about a factor of 2 (as much as a factor of 5 for some programs) over the current highest performance sequential Prolog implementation, the VLSI-BAM. The speedups over other parallel Prolog systems are much larger.

Yale N. Patt
Co-Chair

Alvin M. Despain
Co-Chair

Table of Contents

1. Introduction

Due to the potential usefulness of Prolog [14], a logic programming language, there is great interest in developing high performance Prolog systems. Since its invention by Colmerauer in 1973 [15], the execution speed of sequential Prolog has increased by more than a factor of 1000 due to improvements in compiler and processor technology. Many researchers believe that further performance improvements can be achieved by exploiting parallelism in Prolog. In fact, the declarative nature of the language makes it easier for the compiler to expose parallelism in Prolog programs than in programs written in imperative languages such as C or Pascal. Unfortunately, current parallel Prolog systems rarely perform better than the fastest sequential Prolog systems. The disappointing performance of these parallel systems is due to (1) inefficient sequential Prolog execution of each parallel task and (2) large overhead inherent in complex models of parallel execution. It is tempting to assume that the performance of these parallel systems can be improved significantly by addressing problem (1) alone. However, improving the sequential execution speed of each task would result in a larger fraction of the total execution time being spent on parallel execution overhead. Therefore it is necessary to address problem (2) as well. Since the overhead depends on the model of parallel execution, one way to reduce the overhead is to change the model of parallel execution.

The approach that we take in this dissertation is to develop a model of parallel execution that can be exploited with low overhead and few changes to the efficient sequential execution of each parallel task. We then develop an architecture, called the Flow Parallel Prolog Machine (FPPM), with support for this model of execution as well as support for sequential execution of each parallel task. We refer to the type of parallelism exploited by FPPM as *fine grain parallelism* because the parallel tasks may be small (roughly 20 instructions) and share data frequently with other tasks. Although our model of parallel execution cannot exploit all forms of parallelism that are available to the more complex models, it achieves better performance in most instances and rarely performs worse than sequential execution. Further improvements in performance may be achieved by exploiting other forms of parallelism.

In this chapter, we describe in greater detail the significance of our research problem as well as our research goals. Chapter 2 is a survey of previous work in sequential and parallel execution of logic programs. Chapter 3 describes the type of parallelism that we exploit and explains the overhead associated with the more complex forms of parallelism. Chapter 4 describes the FPPM architecture. Chapter 5 describes an implementation of FPPM. We evaluate the performance of the FPPM implementation and the design tradeoffs in chapter 6. We conclude in chapter 7 with a summary of results and contributions.

1.1. Parallelism in Prolog

There are many forms of parallelism in Prolog [61]. We describe the main forms below (they are discussed in greater detail in chapter 2).

- AND-parallelism is exploited when two or more goals of a clause are executed in parallel. For example, in figure 1.1 if the two goals `goal1(Arg1, Arg3)` and `goal2(Arg2, Arg3)` are executed in parallel then AND-parallelism is exploited.

```
predicate(Arg1, Arg2) :- goal1(Arg1, Arg3), goal2(Arg2, Arg3).
predicate(Arg1, Arg2) :- goal3(Arg1), goal4(Arg2).

?- predicate(12, [1, 2]).
```

Figure 1.1: Example of a predicate and a query.

- OR-parallelism is exploited when multiple clauses of a predicate are executed in parallel. For example, in figure 1.1 if the clauses for `predicate/2` are tried in parallel then OR-parallelism is exploited.
- Unification parallelism is exploited when a goal is unified with a clause head by two or more unification operations executing simultaneously. For example, in figure 1.1 unification of the query, `predicate(12, [1, 2])`, with a clause head, `predicate(Arg1, Arg2)` can be done with two parallel operations, one for each argument. We show in chapter 3 that unification parallelism is a special form of AND-parallelism.

- Stream parallelism is exploited when goals pass a "stream" of alternative variable bindings to other goals; goals that produce the bindings are called *producers* and goals that operate on these bindings are called *consumers*. Stream parallelism may appear to be a new type of concurrency, but it is subsumed by OR-parallelism [29].
- Procedure pipelining is exploited when the execution of consecutive procedures is overlapped (see [4] for an example).
- Prolog execution also involves executing a number of bookkeeping tasks (e.g., choicepoint creation, backtracking). Bookkeeping parallelism is exploited when these tasks are executed in parallel with each other and with unification. Unification and bookkeeping parallelism are finer grain forms of parallelism than either AND- or OR-parallelism.

Most research efforts so far have concentrated on exploiting coarse grain forms of AND-parallelism and OR-parallelism. Unfortunately, none of these projects have shown speedup over the fastest sequential systems without explicit control of parallel execution by the programmer. From the published performance results of parallel Prolog systems [3, 10, 29, 39, 44, 45, 57], we make the following observations:

(1) The benefits of parallel execution are quickly lost due to process creation and bookkeeping overhead. Parallel execution may achieve good speedup over an inefficient sequential implementation because the overhead of parallel execution is small in comparison with the longer sequential execution times. Speedups over good sequential implementations are much harder to achieve. We discuss this issue in greater detail in chapter 2.

(2) Memory management is a hard problem. The parallel processes are usually allocated their own stacks. When the number of processes increases the address space is quickly filled up if the stack space allocated to each process is large. If the stack space allocated to each process is small, stack overflow must be handled. OR-parallelism has the additional complication of having to maintain separate binding environments. Allocating a stack per processor rather than a stack per process alleviates the problem to some extent, but creates new problems due to the fact that stack frames of different processes can be interleaved on a stack. Garbage

collection for such systems is yet another hard problem.

We hypothesize that by exploiting only those forms of fine grain parallelism in Prolog that do not require complex memory management and task management schemes with large overhead we can achieve good speedup over the fastest sequential implementations.

1.2. Research Goals

The high execution overhead of the complex memory management and task management schemes used in exploiting AND- and OR-parallelism leads to the following questions:

(1) What forms of parallelism can be effectively exploited with memory management and task management schemes that have low overhead?

(2) How much speedup can be obtained by exploiting these forms of parallelism?

(3) What architectural support is required to exploit these forms of parallelism?

The goals of this research are to provide answers to these questions, to design a Prolog system that exploits parallelism in Prolog with low overhead memory management and task management schemes, and to demonstrate by means of detailed simulations that such a Prolog system can indeed achieve a significant speedup over the fastest sequential Prolog systems.

We achieve these goals by first identifying the large sources of overhead in parallel Prolog execution: side-effects† caused by parallel tasks, choicepoints created by parallel tasks, task creation, task scheduling, task suspension and context switching.

We then develop a model of parallel execution that eliminates or reduces much of this overhead. Our model of parallel execution restricts parallelism to goals that do not cause side-effects and do not create choicepoints. We refer to this form of parallelism as *flow parallelism*. We eliminate the overhead due to task suspension and context switching by developing program partitioning and task scheduling techniques that exploit flow parallelism without the need for task suspension and context switching. We provide special architectural support to reduce the overhead of task

† Side-effects are actions whose effects remain should backtracking occur. Examples of side-effects are input/output, asserts and retracts.

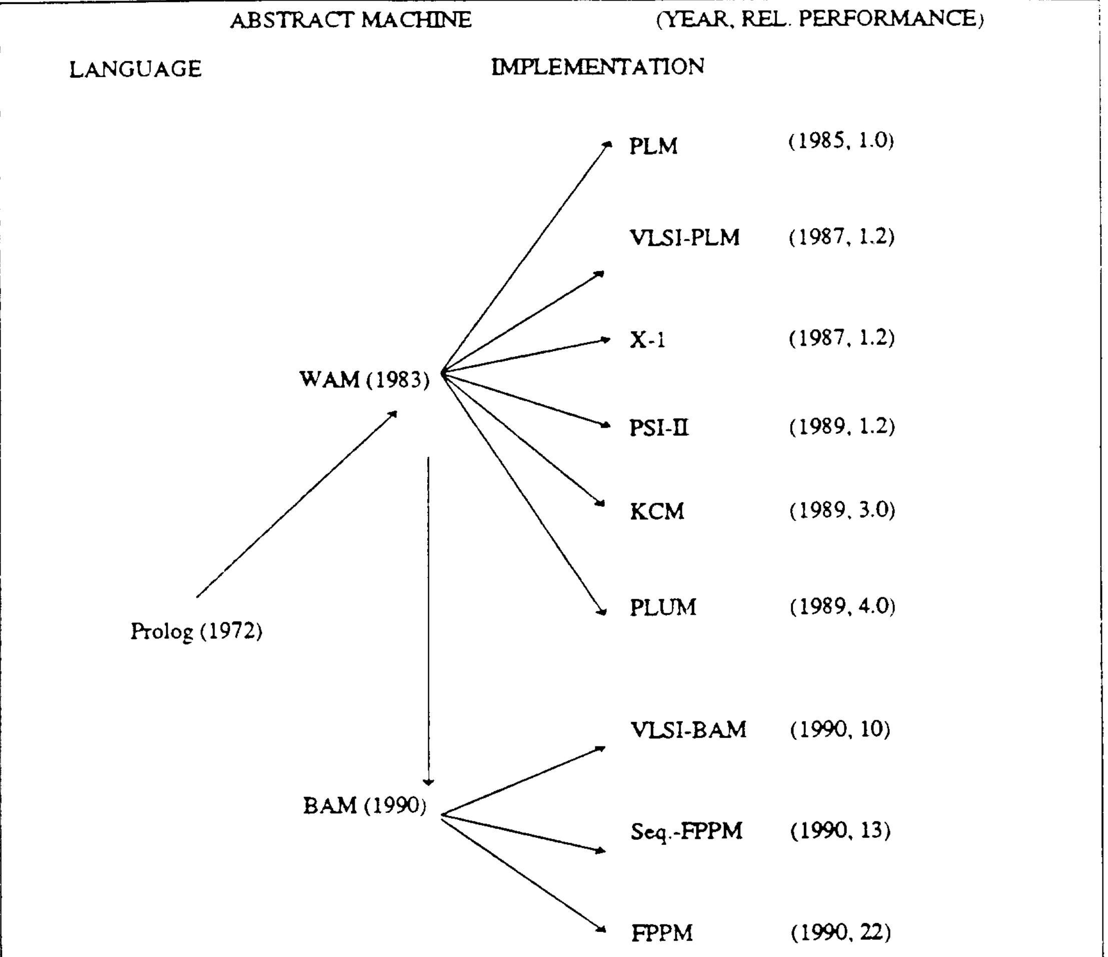


Figure 1.2: Relationship of FPPM to other high performance Prolog machines [5, 22, 24, 25, 40, 49, 59, 60]. The performance of various machines are relative to the performance of the PLM. Cycle time for PLUM is assumed to be the same as the PLM (100 ns.). FPPM's cycle time is assumed to be the same as the VLSI-BAM (33 ns.) and the sequential FPPM code is hand-optimized to include optimizations that could reasonably be expected of a good compiler. Several such optimizations are currently being added to the BAM compiler and VLSI-BAM performance should approach sequential FPPM performance. Parallel code for FPPM generated by hand.

creation and scheduling.

Finally, we demonstrate that our model of parallel execution can be implemented with low overhead memory management and task management schemes by designing and simulating the Flow Parallel Prolog Machine (FPPM). Measurements on a detailed, register-transfer level simulator of FPPM demonstrate that FPPM achieves an average speedup of about a factor of 2 (as much as a factor of 5 for some programs) over the current highest performance sequential Prolog implementation, the VLSI-BAM [40]. The speedups over other Prolog systems are much larger. Figure 1.2

illustrates the relationship of FPPM to high performance Prolog systems based on WAM and BAM. The figure includes the year of introduction for each processor along with its approximate performance relative to the Berkeley PLM [22, 25]. More discussion on these systems is presented in chapter 2.

1.3. Thesis Statement

This dissertation examines the following thesis:

Thesis Statement:
Fine grain forms of parallelism in Prolog can be exploited with low overhead memory management and task management to achieve a speedup of about a factor of 2 over the most efficient sequential implementations.

1.4. Contributions

In the process of proving the thesis statement, this dissertation makes the following contributions:

(1) The identification of sources of large overhead in parallel Prolog execution. These include side-effects, choicepoints and context switching.

(2) A model of parallel execution that restricts parallel execution to those forms of fine grain parallelism that avoid this overhead. This model can be implemented with low overhead memory management and task management schemes.

(3) The design of an architecture, FPPM, with a Main Processor and multiple tightly-coupled Slave Processors that implements this model.

(4) A detailed register transfer level simulator for FPPM. The simulator takes into account execution overhead including pipeline delays and memory latency due to cache misses and cache coherence traffic.

(5) An evaluation of the architecture including the speedups that can be achieved and the investigation of the tradeoffs in the design of the processor.

2. Survey of Previous Research

We survey previous research on high performance sequential Prolog execution in section 2.1. Such a study is important because we want to use the most efficient sequential system as a component of our parallel system. Since the main purpose of parallel execution is to achieve high performance, it is necessary to compare the performance of a parallel system to the most efficient sequential system.

Several parallel Prolog systems have been designed. Section 2.2 is a critique of these systems.

Prolog predicates are often type polymorphic (i.e., they work on arguments of different types). However, in practice many predicates are called with arguments instantiated to terms of the same type each time. Therefore, if one can determine the instantiation of a predicate's arguments at compile time, then the generated code for the predicate can be optimized for those argument instantiations. In section 2.3, we discuss the compile time flow analysis techniques that can be used for compiler optimization. The results of flow analysis are also helpful for parallel execution.

2.1. Sequential Prolog Execution

Early Prolog implementations were interpreters. Prolog compilers provide more than an order of magnitude improvement in performance over the interpreters. David Warren wrote the first Prolog compiler using an intermediate language suitable for Prolog. Later, he refined the intermediate language to the well-known Warren Abstract Machine (WAM) [71], which has since become the basis for the intermediate language of many compilers as well as the basis for the instruction sets of several specialized Prolog architectures. Pipelined special purpose Prolog machines are currently the highest performance Prolog systems available. Until recently, these were based mostly on the WAM. However, a processor based on the new Berkeley Abstract Machine (BAM) [40], together with a new optimizing compiler, has demonstrated much higher performance. The BAM's data structures and memory organization are strongly influenced by the WAM, but its instruction set is simpler and more amenable to compiler optimization. The FPPM's processor instruction set is based on the BAM, but includes some enhancements.

We describe the WAM in greater detail in section 2.1.1. We survey various Prolog systems, both special purpose and general purpose, that are based on the WAM in section 2.1.2. The BAM and an implementation of BAM are described in section 2.1.3. The optimizing compiler for BAM uses flow analysis based on abstract interpretation to obtain static information, such as types and variable instantiation. We describe compiler and flow analysis techniques in section 2.1.4.

2.1.1. The Warren Abstract Machine (WAM)

The WAM specifies data types, data storage areas and an instruction set that is specialized for Prolog execution. Data words contain a type tag field as well as a value field. Since logical variables in Prolog can be bound to data of any type this organization of data words is useful because it allows the type and the value to be determined by examining a single word. This is not a new idea; it has long been used in implementations of Lisp.

The WAM has five data storage areas: choicepoint stack, trail stack, environment stack, heap (also called the global stack) and push down list. The environment stack is similar to the local stack in conventional procedural languages. An environment is used to save register values that must survive a procedure call (each predicate is executed as a procedure). Prolog requires the choicepoint stack and trail stack to implement backtracking. Choicepoints contain copies of the various stack pointers and argument registers at the time the choicepoint was created. These registers are restored from the choicepoint during backtracking. In some implementations, the choicepoints and environments are created on a single stack (also called the local stack). The trail stack is used to record the variables that are bound so that they can be unbound during backtracking. The push down list is a stack used during the unification of nested lists and structures†. The heap is used to store data structures during execution. The heap is managed like a stack; heap space is allocated from the top of the heap. This method is used because heap space can be automatically recovered during backtracking, thus reducing the need for garbage collection. The heap pointer is saved in the

† The push down list is not really required; the environment stack can be used instead.

choicepoint and restored on failure.

WAM consists of six instruction classes: *clause indexing, clause control, procedure control, get, put* and *unify*. The *clause indexing* class contains instructions that filter a set of candidate clause based on an input argument (usually the first). An example is a multi-way branch based on the type tag of the argument. The *clause control* class includes instructions to allocate and deallocate environments and to call and return from procedures. The *procedure control* class contains instructions to create and modify choicepoints. The *get* class includes instructions to perform head unification of simple argument terms and to initiate head unification of complex terms (lists and structures). The *put* class contains instructions to load argument registers of clause goals for simple arguments and to initiate the creation of complex argument terms. The *unify* class contains instructions to unify the elements of complex terms initiated by *get* and *put* instructions.

Dobry *et al* constructed a processor, the PLM [22], that implements a modified version of the WAM instruction set. They also studied the memory access characteristics of the WAM and designed a write buffer and a choicepoint buffer that together improve average performance by 21%. Tick [64] has also compiled extensive measurements on memory access behavior of WAM. Since Prolog processors usually have high memory bandwidth requirements these measurements are particularly useful. Among his results are:

(1) Choicepoint memory traffic dominates the data memory accesses accounting for more than 50% of the total traffic. Most of this is due to shallow backtracking.

(2) Environment accesses account for about 20% of the total data memory traffic.

(3) Heap accesses account for a little less than 20% of the total data memory traffic.

(4) The Trail and Push Down List areas each account for less than 5% of the total data memory traffic.

Tick also studied various caching schemes to reduce the memory traffic. He found that choicepoint memory references could be easily cached with a small, top-of-stack cache because most of the references were due to shallow backtracking. Environment references showed less locality, but could also be cached quite effectively. The heap was least amenable to caching.

Touati and Despain [66] have also studied the dynamic characteristics of the WAM. Among their results are:

(1) Dereference chains are usually short, but dereferencing occurs often.

(2) A large fraction of the choicepoints created by programs can be avoided by compiler optimizations.

(3) Cdr-coding is generally ineffective for Prolog programs.

Although these empirical evaluations of the WAM may lead to some optimizations in compilers and memory system design, WAM based systems are generally slower than implementations based on a new abstract machine, the BAM, which is described in section 2.1.3. The primary problem with the WAM is that its high-level instructions do not allow optimizing compilers to optimize for special cases. A second disadvantage is that the WAM does not contain instructions for common operations such as arithmetic. If an implementation does not provide efficient instructions for such operations, serious performance degradation results. One or both of these disadvantages is apparent in the various styles of WAM implementations described below.

2.1.2. WAM-Based Prolog Systems

Until recently, most high performance Prolog systems were based on the WAM. The three most common implementation methods are (1) macro-coded WAM, (2) threaded WAM code and (3) Micro-coded WAM. These three approaches are discussed in the following sub-sections.

Macro-coded WAM

Prolog programs are compiled into WAM code which is then macro-expanded into the instruction set of the processor. Borriello *et al* [7] have used this approach for the SPUR processor. Mills [46] has suggested a new Prolog architecture to support this technique. Mulder and Tick [48], and Chen and Patt [51] have separately compared execution of the PLM with macro-coded execution of WAM on the Motorola 68020. Macro-expansion of WAM is a useful implementation technique for general purpose processors. The resulting code is quite fast, but with most processors the code size increases dramatically. For example, the static code size for the SPUR was 14 times larger than the equivalent program for the PLM (geometric mean for a set of 15 small benchmarks). Macro expanded WAM on SPUR executes 16 times more instructions than the PLM and requires 2.3 times as many cycles to execute (assuming an ideal memory system). To have the same cache miss ratio, the SPUR implementation requires a cache that is 4 to 8 times size of the PLM's cache.

Threaded WAM code

The Prolog program is compiled into WAM-like code which is then executed by a threaded code interpreter. At least one well-known commercial system, Quintus Prolog [53], uses this approach. The code size for this method is the same as that of the WAM, but the execution speed is slightly slower, in general, than macro-expanded WAM implementations. As with the macro-coded WAM, this method is applicable across a wide range of processor architectures.

Micro-coded WAM

WAM has been implemented through microcode on general-purpose microprogrammable hosts. For example, Gee *et al* [34] added the WAM instructions to the VAX-8600 processor with additional micro-code. This approach is faster than either threaded WAM code or macro-coded WAM. However, it is only applicable to microprogrammable hosts and requires detailed understanding of host microarchitecture.

The highest performance WAM processors are micro-coded implementations with data and control paths designed specifically for WAM. The Berkeley Programmed Logic Machine (PLM) [22, 23, 25] was the first such architecture. The first implementation of the PLM used off-the-shelf TTL components and was the precursor to the Xenologic X-1, as well as a VLSI version built at the University of California, Berkeley [60]. Although the PLM achieves between 100 and 300 KLIPS (Kilo Logical Inferences per Second) on some benchmarks, it relies on a general purpose host processor to execute built-in functions. This seriously degrades performance in most programs. The ECRC Knowledge Crunching Machine (KCM) [5] and the Japanese PSI-II [49] are other examples of WAM-based processors that have been constructed. The KCM achieves about 3 times the performance of the PLM with a 64-bit instruction format that is capable of two register transfers per cycle, support for arithmetic operations unification and backtracking, and a faster cycle time (80 nsec.). The PSI-II achieves slightly higher performance than the PLM and is designed as part of a multiprocessor to execute the KL1 language.

2.1.3. The Berkeley Abstract Machine (BAM)

The WAM has been instrumental in the development of Prolog compiler technology. However, although the data structures and memory organization of the WAM are very well suited to Prolog, its instruction set precludes many compiler optimizations. The Berkeley Abstract Machine (BAM) [40] employs many aspects of the data structures and memory organizations of the WAM, but BAM's instruction set is more general-purpose, consists of more primitive operations and has additional optimized operations. This permits greater optimization and flexibility in code generation. The BAM has some support for Prolog operations such as multi-way branches, type tags and dereference instructions. Extensive simulation results show that a processor based on the BAM, the VLSI-BAM achieves a speedup of about a factor of 10 over the PLM. This factor of 10 speedup consists of a factor of about 3 due to increased clock frequency (30MHz for VLSI-BAM versus 10 MHz for the VLSI-PLM), and a factor of about 3 due to compiler and architecture optimizations. The special features of the VLSI-BAM for Prolog execution use about 11% of the VLSI-BAM's chip area, but result in about 70% speedup [40].

The instruction set of FPPM processors and Prolog compilation for sequential execution borrows extensively from the BAM.

2.2. Parallel Prolog Execution

Although the fastest Prolog systems currently available are sequential, there is potential for higher performance through parallel execution. Prolog was originally designed for sequential execution and its semantics often imposes sequentiality on program execution. In an effort to simplify the task of parallel execution of logic programs several researchers designed new languages called concurrent (or committed choice) logic programming languages. We review these languages in section 2.2.1. Although these languages simplify the problem of exploiting parallelism, they do so at the expense of a degradation of the logic programming paradigm. There have been several efforts to exploit parallelism in Prolog while maintaining its sequential semantics. Most of these efforts have concentrated on AND-parallelism (parallel execution of goals in a clause) and OR-parallelism (parallel execution of the clauses in a predicate). We survey AND- and OR-parallel systems in section 2.2.2. There have also been efforts to exploit parallelism in Prolog at a finer granularity than AND- or OR-parallel processes. In section 2.2.3 we survey attempts to exploit fine grain forms of parallelism in Prolog.

In our reviews of several models of parallel execution, we consider the following questions carefully:

(1) **How are the parallel tasks created?** Since there is an overhead for parallel task creation, only tasks that are large enough should be executed in parallel. In general, it is hard to automatically determine which tasks are large enough to execute in parallel. The system can use heuristics to estimate the size of a task, but since such estimates are often inaccurate, it is essential for such a system to have low overhead for parallel execution so that the performance penalty for creating too small a task is acceptable. The overhead is less crucial (but is still important) if the programmer explicitly controls parallel task creation in his/her program.

(2) **How is the model of parallel execution evaluated?** Designers often use inefficient sequential systems (hardware or software) as components in their parallel system (usually because these are easier to obtain or to modify for parallel execution). Although these systems obtain good speedups over the inefficient sequential components, they are often slower than the most efficient sequential systems. There is then a temptation to extrapolate their speedup to estimate what it would have been had more efficient sequential components been used instead. Unfortunately, the interactions among the various aspects of a parallel computer system are so complex that it is almost impossible to obtain accurate extrapolations. This is especially true if the two sequential components differ in performance by more than an order of magnitude.

The overhead of parallel execution is particularly important in this regard. If the overhead is mainly a function of the model of parallel execution, rather than the particular sequential component used to implement the model, then the overhead will not change if the sequential component is replaced by a more efficient one. As the performance of sequential component is increased, the speedup is limited by the fixed overhead of the parallel execution model.

2.2.1. Concurrent (Committed Choice) Logic Programming Languages

A number of *concurrent logic programming languages* (also called *committed choice logic programming languages*) have been designed specifically for parallel execution. In these languages each goal is executed by a separate process and shared variables represent streams for inter-process communication. The the three principle committed choice languages are Concurrent Prolog [27], Parlog [36] and Guarded Horn Clauses (GHC) [41]. The differences between these languages is not pertinent to this research. We discuss only the most important characteristics of these languages in this section (see [56] for a more complete survey). A concurrent logic program is a collection of guarded clauses. A guarded clause differs from a Prolog clause in that the goals in the body of the clause that appear before the *commit operator*, " | ", constitute the *guard* and are treated differently from the remaining goals. All the candidate clauses for a goal may execute in parallel until the commit operator. At the commit operator only one clause may continue execution and all others are discarded. Committing to one clause simplifies parallel execution, but degrades the logic programming paradigm.

The guards of concurrent logic programming languages could be user defined predicates that invoke the execution of other clauses. Thus, execution of the guards could result in large numbers of clauses being simultaneously executed. This is difficult to implement efficiently. In the *flat* versions of each of the languages the guards may not be arbitrary goals, but rather must belong to a predefined set of deterministic predicates. Currently, most research on concurrent logic programming languages concentrates on the flat languages because they are simpler to implement.

The essential difference between Prolog and the concurrent logic programming languages is that Prolog is *non-deterministic* (also called *don't know* non-deterministic) whereas the concurrent logic programming languages are *in-deterministic* (also called *don't care* non-deterministic). In simpler terms, Prolog searches the space of solutions without discarding alternate paths to solutions, whereas concurrent logic programming languages discard alternate paths after selecting one path to explore. Although committed choice languages may simplify the task of parallel execution, they do so at the expense of a significant degradation of the logic programming paradigm.

There are a number of implementations of committed choice languages. Ginosar and Harsat [37] describe Carmel, a sequential processor for a variant of Flat Concurrent Prolog. Alkalaj and Shapiro [1] describe an architecture that consists of several specialized processing units that exploit internal (fine grain) concurrency and a specialized memory hierarchy. Both the processors above are uniprocessors. There are also a number of parallel processor implementations, both shared memory and distributed. Taylor *et al* [62] describe a distributed implementation of Flat Concurrent Prolog (FCP), Ichiyoshi *et al* [41] describe a distributed implementation of Flat Guarded Horn Clauses (FGHC), Foster [30] describes a distributed implementation of Flat Parlog. Crammond [18] describes a shared memory implementation of Parlog. *Strand*, [31] a new language that uses simple assignment instead of general unification to instantiate shared variables, has commercial implementations on a number of parallel architectures including both distributed and shared memory architectures.

The performance of concurrent logic programming language implementations has been disappointing and there are few performance comparisons with efficient sequential systems. The overhead of process management and memory management for parallel execution are quite high compared to the amount of useful computation. Systems are also often swamped by large numbers of processes, resulting in large overhead for context switching and scheduling.

In FPPM, we use shared variables to represent flow of data from one task to the next. However, instead of suspending a process and switching contexts to another process when no data is available, FPPM's task busy-waits for data. We explain how this is achieved without the danger of deadlock in chapter 3.

2.2.2. AND-parallelism and OR-parallelism

Rather than define new languages specifically for parallel execution there has been considerable effort in trying to exploit parallelism in Prolog while maintaining the semantics of sequential Prolog. Most of this research has concentrated on AND- and OR-parallelism. Some systems, such as the RAP-WAM [19] and APEX [44], exploit only AND-parallelism. Other systems, such as the Aurora system [45], exploit only OR-parallelism. Yet other systems, such as PPP [29], PEPSys [72] and Conery's work [16], exploit both AND- and OR-parallelism.

Unfortunately, none of these systems outperform the fastest sequential implementations, although many have demonstrated speedups over single processor implementations of their own system. The main reasons for this are (1) the overhead associated with parallel execution is too high, especially when the granularity of the parallel task is small, and (2) sequential execution on each processor is inefficient. With improving Prolog compiler technology, such as Van Roy's [69], the speed of the sequential Prolog execution component of these systems can be expected to increase by at least a factor of 5 over the current highest performance sequential systems, whereas it is unlikely that the execution overhead will decrease significantly unless the model of execution is changed. Therefore the fraction of execution time spent on overhead for parallel execution will increase rapidly. It is essential, therefore, to reduce the overhead associated with parallel execution if significant performance improvements are to be achieved. In this section, we review parallel Prolog systems and identify sources of overhead in each of them.

Types of AND-parallel Systems and their Overhead

One of the important issues when exploiting AND-parallelism is dealing with shared variables. Shared variables are unbound variables that occur (at run time) in the arguments of more than one goal in the body of a clause. Goals that share variables cannot, in general, be executed in parallel because they may bind the shared variable to different terms. Almost all AND-parallel systems execute goals in parallel only if they do not share uninstantiated variables. A few systems, such as Conery's, determine at run time whether a variable is instantiated. The overhead involved in such tests is likely to be quite substantial (Conery does not provide estimates of the overhead).

Some systems rely on static analysis of the program to identify independent goals. An example of such a system is the PPP [29] which uses Static Data Dependency Analysis (SDDA) proposed by Chang, Despain and DeGroot [11] to identify independent goals that may be executed in parallel. SDDA is an instance of a more general technique called abstract interpretation. Static analysis using abstract interpretation plays a very important role in this dissertation and we discuss it in greater detail in section 2.3. DeGroot's Restricted AND-Parallel WAM (RAP-WAM) [19] uses simple run-time tests called *conditional graph expressions* (CGEs) to identify independent goals. If parallelism is exploited for independent goals only then the amount of parallelism is limited. AND-parallel systems that allow parallel tasks to create choicepoints incur overhead due to a number of reasons that are described in chapter 3.

Types of OR-parallel Systems and their Overhead

There are several variations in OR-parallelism:

- Single-solution OR-parallelism executes multiple clauses in the search for alternative solutions, but only one solution is produced. The time spent computing other solutions is wasted. The execution time depends on the Prolog semantics that the system must follow. For conventional Prolog semantics the solutions must be presented in the order of the clauses in the program. Faster execution is possible when the semantics allow the solutions to be presented in any order.
- All-solutions OR-parallelism produces all the solutions to a query. This type of system wastes computing resources if all the solutions are not required.
- In OR-parallelism with continuations, a process is created for each alternative *path* that the computation can take; each process solves not only the goal, but also the remaining goals to be executed (i.e., the continuation). It has been shown [29] that an OR-parallelism with continuations subsumes stream parallelism.
- In OR-parallelism without continuations, a process is created for each alternative solution to a *goal*. This form of OR-parallelism cannot exploit stream parallelism.

An important issue with OR-parallelism is how to maintain independent binding environments so that the bindings for variables made by one clause do not affect the execution of other clauses that are tried in parallel. A simple solution is to create separate copies of the arguments for each clause that is tried in parallel. This method is used by Kale *et al* in the Reduce-OR system [57]. This method is particularly attractive for distributed memory architectures. However, the overhead of making copies of arguments can dominate the execution time. Some of this overhead can be eliminated by copying only the unbound variables into separate binding environments. The PPP [29] uses *hash windows* of variables to maintain separate binding environments. The hash windows are linked to form a tree. One problem with the scheme is that an OR-parallel process may have to traverse a chain from a node to the root of the tree of hash windows to look for a binding for a variable. Another problem is the inherent overhead of maintaining the hash tables. The Aurora system [45] uses *binding arrays* instead of hash windows to maintain multiple binding environments. A binding array contains copies of all potentially shared variables. The advantage of binding arrays over hash windows is that there is no need to traverse a chain of binding arrays. The disadvantage of binding arrays is that a new binding array has to be created and initialized for each new process, thus increasing the cost of process creation. The PEPSys [3] system uses a combination of time stamps and hash windows. All but one of the OR-branches have hash windows. When a variable is bound, the OR-branch "level" is used as a time stamp for the binding. When a choicepoint is created, the OR-branch "level" is incremented. OR-processes can trust bindings with time stamps earlier than the process time stamp (the "level" when the process was created). This method allows relatively low-cost process creation, but binding and dereferencing have additional overhead due to time stamps.

The various parallel Prolog systems also differ in the techniques that they use for process management, memory management and inter-process communication. Process management includes creation, scheduling and termination of processes. A detailed analysis of the complex tradeoffs with each of these activities is outside the scope of this dissertation, but some of the issues that must be considered are as follows. The overhead of process creation is quite high and may dominate the total execution time if the processes created perform only small computations. Parallel execution in Prolog is often speculative; if speculative processes are scheduled ahead of non-speculative processes, then the processors could be performing unnecessary computations. In search-intensive applications, the number of processes created increases combinatorially and could swamp the system. When a speculative process is terminated because it has been determined that its results are no longer relevant to the final result, the descendants of the process must also be terminated. However, the descendants also create processes, possibly faster than the processes are killed. Thus unnecessary processes could proliferate on the system.

There are essentially two approaches to memory management for AND- and OR-parallel Prolog architectures. One allocates stacks for each process and the other allocates stacks for each processor. The PPP is an example of a system that uses the former and the RAP-WAM is an example of a system that uses the latter. The disadvantage of allocating separate stacks for each process is that the address space becomes fragmented. If the size of the allocated stacks is large then the address space could be completely used up. On the other hand, if the size of the allocated stacks is small, then frequent stack overflows must be handled. The disadvantage of allocating stacks for each processor is that each stack contains frames corresponding to different processes. These frames could be interleaved on the stack. Consequently, it is hard to resume processes that need to add frames to the stack, and the space for stack frames that are deallocated by a process may not be reclaimed.

Processes in parallel Prolog systems communicate by messages. Since most of these messages must obtain locks to manipulate global data structures, they may have a substantial overhead that eliminates some of the benefits of parallel execution.

Performance of AND- and OR-parallel systems

There are only a few examples of performance measurements of parallel Prolog in which performance is compared with a fast, sequential Prolog system, such as Quintus Prolog. Most articles on parallel Prolog execution present speedups over single processor systems executing their model of parallel execution. The sequential code that is executed on each processor is also often very inefficient. It almost impossible to accurately extrapolate from these measurements to obtain speed-ups over a good sequential implementation if the processors used efficient sequential code. A few examples where the parallel execution speed has been compared to good sequential Prolog implementations are: the Aurora OR-parallel system [45], PEPSys [3], APEX [44] and Reduce-OR [57]. In each of these cases, the parallel performance is rarely better than Quintus Prolog on a Sun 3/50. It is also important to note that all of the systems, except for Aurora, rely on programmer annotations to create tasks only for computations that are large enough. Without these annotations the performance would be worse. We believe that such compiler annotations undermine the benefits of using Prolog for parallel execution. Automatic generation of such annotations is as yet an unsolved problem.

We illustrate the poor performance of parallel Prolog systems in comparison with the fastest current sequential Prolog systems by considering the following examples:

(1) The Aurora [45] OR-parallel system is, to the best of our knowledge, among the most successful parallel Prolog systems in terms of performance. Aurora has modified Sicstus Prolog (a sequential Prolog system) for parallel execution. Executing on an Encore Multimax multiprocessor they report a speedup (over their own system running on a single processor of the Multimax) of between 5.8 and 14.2 with 16 processors. The overhead of parallel execution is approximately 25%. This overhead is caused mainly by (1) task switching (including updating binding arrays) and (2) synchronization and locking. Neither of these causes of overhead is likely to improve significantly if the sequential component (Sicstus Prolog) is replaced by a more efficient Prolog system. Quintus Prolog, for example, is approximately 2 times faster than Sicstus. More recently, Taylor's compiler has demonstrated approximately 24 times (geometric mean) the performance of Sicstus on the same processor [63]. Unless the overhead of Aurora's parallel execution model is reduced significantly, its performance will not improve by much if Sicstus is replaced by Taylor's compiler; in fact, parallel execution may be slower than sequential execution.

(2) The APEX [44] system exploits AND parallelism on a Sequent Balance 21000. The system

achieves speedups of up to 17.9 (over one of their own processors) using 20 processors. The sequential execution component, a byte code WAM interpreter, is approximately 5 times slower than Quintus Prolog on the same processor (or roughly 60 times slower than Taylor's compiler on the same processor). The overhead for parallel execution is small, only between 1% and 2% of sequential execution time. However, the overhead is so small primarily because the programs chosen explicitly control the granularity of parallel tasks; a task is not executed in parallel if it is small compared to the overhead of parallel execution. Without such explicit control, the overhead is much greater and the speedup is smaller. For a benchmark for which no granularity control was done (`quicksort` of a 511 element list chosen so that the execution tree is balanced), the overhead was 12% and the speedup was only 3.2.

2.2.3. Fine Grain Parallelism in Prolog

The granularity and execution overhead of parallel processes in AND- and OR-parallel systems is quite large. There are other forms of parallelism with finer grain size and lower overhead. In this section we review techniques and architectures to exploit fine grain parallelism in Prolog.

2.2.3.1. Wide Instruction Word Architectures

One approach is to exploit is to exploit parallelism at the level of micro-operations using wide instruction word that can issue multiple operations per cycle. Very Long Instruction Word (VLIW) architectures combined with trace scheduling compilers have been able to schedule several operations per cycle for numerical codes that have regular structures [28]. The Cydra directed data flow architecture [54] was also able to statically schedule multiple operations per cycle for numerical codes. However, static scheduling of multiple operations per cycle in a single instruction word is more difficult for Prolog programs that tend to be less regular. Carlson [9] has been able to achieve a speedup of about a factor of 2 with loop unrolling and trace scheduling on an idealized wide instruction word architecture. He plots performance as a function of the available resources for a number of benchmarks and concludes that the processor could achieve most of the speedup with 2 data memory ports and 3 ALUs. It is important to note that Carlson's speedup measures are based on an idealized machine model in which all instructions complete in one cycle, including indexed memory loads and conditional branches. Since code sequences that include memory loads followed by conditional branches are common in Prolog programs, it is very likely that the speedup will be much lower with more realistic assumptions about instruction latencies.

2.2.3.2. Unification Parallelism and Bookkeeping Parallelism

Another approach to exploiting fine grain parallelism in Prolog is to identify special types of tasks that can be executed in parallel on separate processors or function units. Bookkeeping tasks (creating and restoring choicepoints and environments, trailing variables and clause indexing) are examples. Another example is unification. Unification is the process of making two terms identical, if possible, by finding substitutions for some or all of the variables in the terms. If the terms are complex (i.e. lists or structures) then each pair of the corresponding arguments of the term are unified recursively. In Prolog, unification is a basic operation that is done at each goal invocation (which corresponds to a procedure call in an imperative language) to match goals with clause heads. Parallelism in the unification of two complex terms can be exploited by unifying the corresponding arguments of the terms in parallel. However, if a variable occurs in two parallel unifications then a data dependency exists between them because the two unifications must bind the variable to the same term. Such data dependencies limit the amount of parallelism in unification. In fact, the theoretical results on parallel unification described below imply that in the worst case unification is not amenable to parallelism. Dwork *et al* [26] and Yasuura [73] have independently shown that unifiability is log space complete for **P**, where **P** is the class of problems that can be solved sequentially in time bounded by a polynomial of the input size. This means that unification is in the class NC (NC is the class of problems that can be solved in $O(\log^k I)$ time for some constant k where I is the input size and the number of processors, P, is bounded by a polynomial of I) only if **P** = NC, which is considered highly unlikely. This implies that it is highly unlikely that unification can be computed in $O(\log^k n)$ parallel time using polynomial number of processors. Prolog's unification is a special case of general unification because the goal and the clause head being unified do not share variables. However, Citrin [13] has shown that Prolog head unification is also log space complete for P by constructing a log space, linear time reduction from general unification to Prolog head unification.

The results above indicate that general unification cannot achieve good speedup due to parallelism. However, Vitter and Simmons [70] showed that unification can run in $O(\frac{E}{P}+V\log P)$ where P is the number of processors, E the number of edges in the term graph and V the number of vertices. Thus, although general unification is not amenable to parallelism, there are many instances of unification that can benefit from parallel execution. Indeed, it is easy to see that unification of two complex terms whose arguments do not contain shared variables can benefit from parallelism. The corresponding arguments can be unified in parallel.

Any architecture that attempts to exploit unification parallelism must address the problems of identifying and handling shared variables. The problem of identifying shared variables in unification is almost identical to that of identifying shared variables in Prolog goals for AND-parallelism described earlier. One approach, proposed by Citrin [13], is to generate a schedule of unifications at compile time such that all the unifications that execute in parallel are independent. The schedule is generated using a static data dependency analysis (SDDA) derived from that of Chang *et al* [11]. As mentioned earlier, SDDA is an instance of abstract interpretation (see section 2.3.). In Citrin's scheme, a schedule consists of one or more sets, each set consisting of unifications that may execute in parallel. A set is allowed to execute only after all the unifications in the previous set are complete. This scheme has the advantage of simplicity because the parallel unifications are always independent. However, the amount of speedup is limited because (1) the unifications belonging to different sets may not overlap, (2) the execution time of a set is the execution time of the longest unification in the set, (3) unifications belonging to different goal executions may not overlap, and (4) static schedules reflect the worst-case assumptions about shared variables. Citrin observed that on an average each set of unifications contain 3 unifications. However, this measure is not very useful for estimating performance improvement due to unification parallelism because execution time of unifications, overhead associated with parallel execution and time taken for book-keeping operations are not considered.

In the Parallel Unification Processor (PUP), we took a different approach to exploiting unification and bookkeeping parallelism [12]. Run time synchronization was used to handle data dependencies (including shared variables) and restricted data flow techniques of HPS [50] were used to allow out-of-order execution. The write-once property of logical variables was used to handle shared variables. Two cases of shared variables were considered:

(1) The variable appears as explicitly shared in the code. In this case a *write-once* register was allocated to the variable. Only one unification is allowed to bind the variable by writing to the register. Once the register is written a valid bit associated with the register is set and no other writes to the register are allowed. If other unifications attempt to unify with the variable they read the value of the register instead of attempting to bind it.

(2) The variable does not appear in the code. In this case unifications must obtain a *dereference lock* before binding any variable. Any unification that attempts to dereference a variable that has been locked must wait until the variable is bound.

PUP executed bookkeeping operations in parallel with unification. Experience with PUP showed that exploiting unification parallelism alone cannot result in significant speedup; bookkeeping tasks must also be executed in parallel. PUP also showed that dynamic scheduling is effective for parallel unification and bookkeeping. The problems with PUP were equally instructive: (1) run time synchronization for all variable bindings contributed to large overhead and (2) latencies associated with dispatching unifications for parallel execution limited parallelism in short loops.

Beer's POPE processor [4] took a novel approach to exploiting fine grain parallelism in Prolog. Rather than exploit parallelism within each goal, POPE pipelines execution of consecutive goals by executing each goal on a separate processor that is part of a circular pipeline as shown in figure 2.1. We term this form of parallelism *procedure pipelining*. Consecutive processors in the pipeline share a register file that acts as the pipeline buffer. Each processor has access to two register sets; the processor obtains its inputs from one and writes its outputs to the other. The outputs of one processor become the inputs for the next processor in the circular pipeline. Each processor executes the code corresponding to the head of a Prolog clause and creates arguments for the next goal

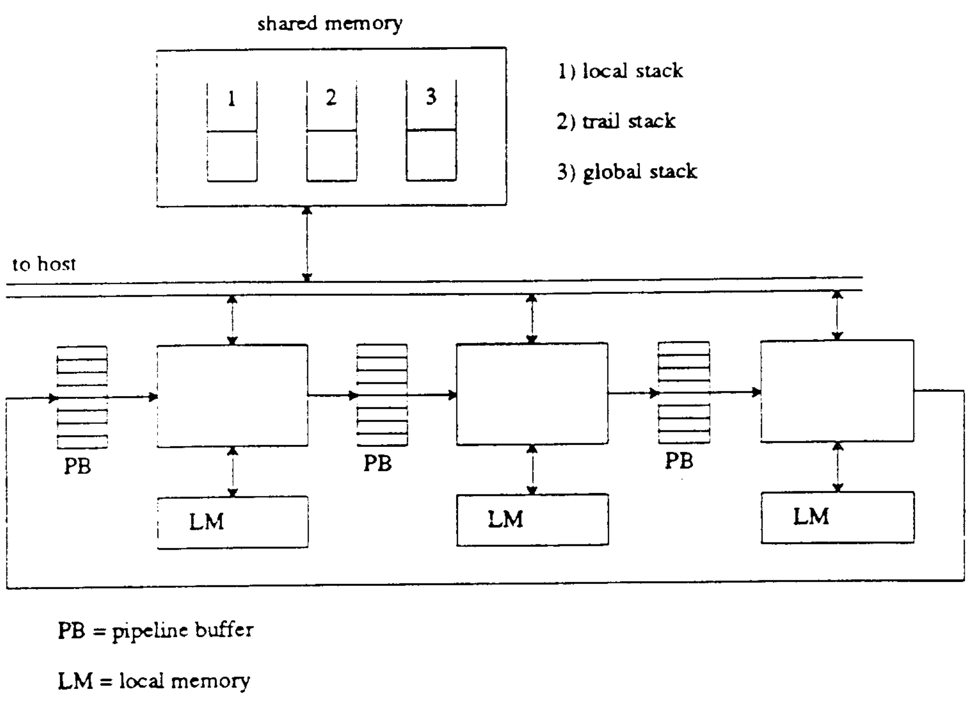


Figure 2.1: Overview of the POPE Processor.

in its output registers (which are the inputs for the subsequent processor in the pipeline). It supplies the address of the code for the next goal to the subsequent processor before writing the the output arguments so that the subsequent processor can begin fetching the code for the procedure and execute bookkeeping operations while waiting for the argument registers to be written. Each register has a "hardware semaphore" that causes a processor to stall if it tries to read it before it has been written. The hardware semaphores are reset when the execution "wraps around" the circular pipeline of processors. The registers in each register set contain all the information required for a choicepoint and form a choicepoint buffer. Conflicts due to variables that could be bound by unifications in multiple procedures are handled by disallowing bindings in a procedure until the trail pointer is valid in its input register set and ensuring that the trail pointer is written to the next register set only after all the bindings in the current procedure are complete. Similarly, conflicts between procedures for heap allocation are handled by allowing heap space allocation only with a valid heap pointer and writing out the heap pointer only when all the required heap space for the current processor has been allocated.

The POPE contributed several important ideas to the design of FPPM, including the multiple register sets to eliminate output dependencies between consecutive goal executions. However, the POPE exploits parallelism only among different procedures; each procedure is executed sequentially.

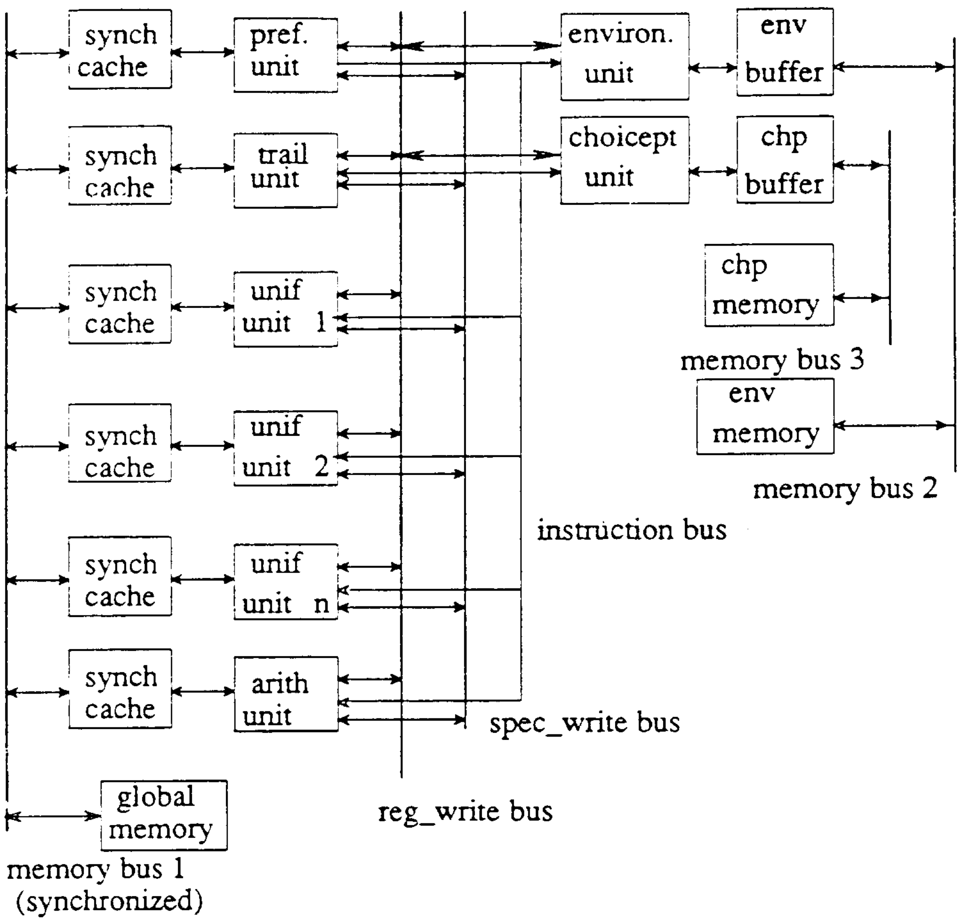


Figure 2.2: Overview of PLUM

In an earlier experiment we explored the speedup due to unification and bookkeeping parallelism in a WAM-based system by designing and simulating the Parallel Unification Machine (PLUM). Figure 2.2 is an overview of this processor. It consists of a Prefetch Unit, an Environment Unit, a Choicepoint Unit, a Trail Unit and multiple Unification Units. The Prefetch Unit fetches instructions from memory and dispatches unification and bookkeeping operations to the appropriate function units. The architecture achieves a speedup of between 3 and 4 over the VLSI-

PLM. Measurements for the processor are described in [58, 59].

benchmark	PLUM cycles	VLSI-PLM cycles	Speedup
fib	12397	30180	2.4
hanoi	8324	24788	3.0
nreverse	7431	21160	2.9
qsort	9720	40322	4.2
queens	2682	6649	2.5
tak	6423	27848	4.3
arith. mean	-	-	3.2
geom. mean	-	-	3.1

Table 2.1: Comparison of PLUM with Berkeley VLSI-PLM

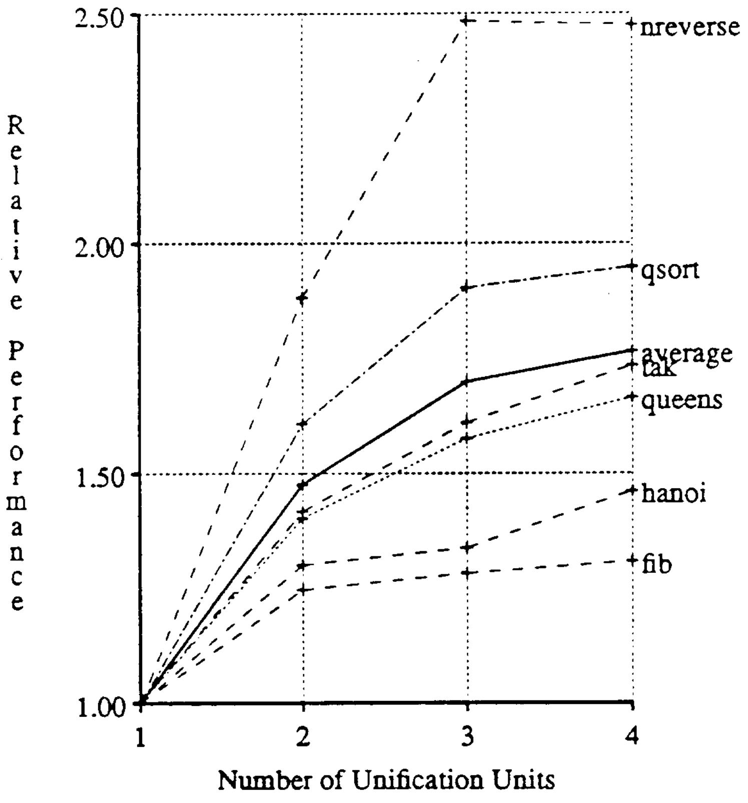


Figure 2.3: Relative performance versus number of Unification Units for PLUM

PLUM achieves a speedup of approximately a factor of 3.2 over a sequential WAM-based processor, the Berkeley VLSI-PLM [60]. The speedup of PLUM over the Berkeley VLSI-PLM is shown in table 2.1 for a set of 6 benchmarks. The effects of multiple Unification Units on relative performance of PLUM is plotted in figure 2.3 (performance is relative to PLUM with 1 Unification

Unit).

2.2.3.3. Data Flow Architectures

Another approach to exploiting fine grain parallelism in Prolog is to use data flow architectures [2, 21]. We discuss data flow processors because we use several techniques that are inspired by data flow in the design of the FPPM. There have also been proposals to construct data flow processors specialized for Prolog execution [38, 42]. To the best of our knowledge, none have actually been built.

Data flow processors can, at least in principle, exploit all the available parallelism in programs. However, they require expensive hardware and substantial communication overhead during program execution [32]. The *macro data flow* model of computation eliminates some of the communication overhead of data flow at the expense of some parallelism. In a macro data flow processor a node of the data flow graph is not a primitive operation (instruction), but rather a collection of such operations called a *task*. The tasks should be chosen such that the operations within the task have closely-knit dependencies and there are relatively few dependencies between operations of different tasks. Each task is executed on a processor with efficient local communication to handle the dependencies within the task. The expensive global communication is used only for the dependencies between tasks. The Cedar multiprocessor being developed at the University of Illinois, Urbana-Champaign [33] is an example of a system that uses a macro data flow model of computation. In Cedar each collection of operations is called a *Compound Function* (CF) and is executed on a cluster of tightly coupled processors. A CF is dispatched for execution on a processor cluster only when all its input operands are available.

A major drawback of the data flow model of computation is its inability to perform computations that involve changing large data structures efficiently [32]. In the classical von Neumann model of computation data structures are manipulated in memory by a sequence of instructions and the simple storage model for data structures is possible only because of the implied sequence of operations. The *restricted data flow* model proposed by Patt *et al* for the High Performance Substrate processor (HPS) [50] exploits parallelism within a sequential instruction stream. In HPS a data flow graph representing the instructions within a window in the sequential instruction stream is dynamically constructed and executed. The program executes by sliding the window along the sequential instruction stream, adding nodes representing new instructions to the data flow graph and retiring instructions when the nodes have been executed. Thus, parallelism within a data flow graph restricted to a window of instructions is exploited using data flow techniques. This approach is, among other things, a generalization of the Tomasulo algorithm used in the floating point processor of the IBM 360/91 [65].

The FPPM architecture is a synthesis of macro data flow and restricted data flow. As in macro data flow, the nodes in the program data flow graph are tasks rather than individual instructions. As in restricted data flow, only a restricted set of nodes in this data flow graph can be scheduled for execution at any time. Consequently, we call the model of computation *restricted macro data flow*.

2.3. Flow Analysis and Optimizing Prolog Compilers

Flow analysis plays an important role in exploiting parallelism and in sequential code optimizations. Prolog predicates are often polymorphic. Without compile time knowledge of the actual instantiations for the predicate's arguments the compiler must generate code that executes the predicate correctly for all possible argument instantiations. In practice, many predicates are called with their arguments instantiated in the same way each time. If the types of the arguments for a predicate are known at compile time, then the generated code can be optimized to handle those types of arguments. Flow analysis is used to obtain this information about predicate arguments at compile time.

We explain in chapter 3 that goals that do not create choicepoints or execute side-effects can be executed as parallel tasks without complex memory management or task management schemes. With flow analysis the compiler can often detect such goals. For example, consider the program below which reverses a list.

```
?- reverse([1,2,3],R).

reverse([],[]).
reverse([X|L1],L):- reverse(L1,L2), append(L2,[X],L).

append([],X,X).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).
```

Flow analysis programs, such as those written by Van Roy [68], Chang *et al* [11] and Citrin [13], can determine that the first arguments of both the `reverse/2` and `append/2` predicates are non-variable terms. Therefore, only one of the two clauses in each predicate can ever succeed and a simple test of the first argument can identify this clause. No choicepoint is ever created for either of the predicates and FPPM could execute them as parallel tasks. Other examples of abstract interpretation for mode and type inferencing are [8, 20].

Automatic flow analysis programs for Prolog are based on a general flow analysis technique called *abstract interpretation*. The mathematical basis for abstract interpretation was first described by Cousot and Cousot [17]. The basic idea in abstract interpretation for mode analysis in Prolog is to transform the program so that the arguments of the predicates in the new program are restricted to a finite *lattice* (i.e., it is partially ordered and every subset has a least upper bound and a greatest lower bound). Each argument in the transformed program represents a superset of the actual values of the corresponding argument of the original program. Each lattice element represents an infinite set of values. Since the lattice is finite, the transformed program can be executed symbolically (in finite time) until a fixed-point is reached. If the conditions of abstract interpretation hold, then the least fixed-point of the program's symbolic execution over the lattice is a conservative approximation to the actual set of values the predicates' arguments can take at run time.

3. Fine Grain Parallelism

In order to exploit parallelism in a program, three problems must be solved:

(1) **Data flow analysis**. This step identifies the operations that may be executed in parallel and determines the total amount of parallelism available.

(2) **Program partitioning**. This step selects the sizes of tasks as well as the amount of communication and synchronization required to satisfy dependencies between tasks.

(3) **Task scheduling**. This step attempts to allocate processors to minimize total execution time.

This chapter describes our approach to each of these problems. In section 3.1, we explain how we use data flow analysis to simplify handling of data dependencies. There are several solutions to the program partitioning and task scheduling problems. In keeping with the goals of this dissertation, we choose low overhead program partitioning and task scheduling techniques, even though they restrict the amount of parallelism that can be exploited. Although more complex techniques can expose additional parallelism, they also incur greater overhead that reduces the performance benefits. We describe our program partitioning and task scheduling methods in section 3.2. An alternative scheme for program partitioning and task task scheduling is described in section 3.3. This alternative exposes more parallelism, but also incurs greater scheduling overhead. We compare the two methods in chapter 6.

Each of the three problems above can be solved statically (at compile time) or dynamically (at run time). The advantage of compile time solutions is that there is no run time overhead. On the other hand, more specific run time information can result in more accurate run time solutions. Data flow analysis and program partitioning are almost always done at compile time because they require a sophisticated analysis that would be too slow at run time. We do the same in this dissertation.

Task scheduling is done either statically or dynamically. If sufficiently detailed information about the task execution times and dependencies is available, then static task scheduling is preferable to dynamic task scheduling. Sarkar [55] has quantified the benefits of static scheduling over dynamic scheduling for predominantly numerical benchmarks written in the single assignment language SISAL. His static scheduling requires complete information about the data flow between tasks as well as detailed profile information such as execution frequencies of tasks. It remains to be seen if his static scheduling methods will be as successful for non-numerical benchmarks in Prolog. In our opinion, dynamic task scheduling has greater promise for non-numerical problems in Prolog because complete profile and data flow information may be harder to obtain due to greater dynamic variability in computation paths, dynamic typing and non-deterministic execution. Consequently, we choose to investigate dynamic task scheduling.

3.1. Handling Data Dependencies with Data Flow Analysis

Information is transferred from one Prolog goal to another through shared variables. In the example below, suppose that `compute1/3` binds `S1` to a value that it computes using `A` and `B`. This value is then transferred to `compute2/3` through the shared variable `S1`.

```
 ?- goal1([1,2,3], Y).

goal1([A, B, C], S) :-
    compute1(A, B, S1),
    compute2(S1, C, S).
```

Variables may be shared even though they appear as different variables in the text of the predicate definition. This is because a variable may be *aliased* (i.e., bound to) to other variables when the predicate is actually called. In the example below, the variables `X` and `Y` appear to be different logical variables. However, the goal `a/2` binds the two variables to each other so that `X` and `Y` are aliased to each other when `b/2` is called.

```
 ?- goal2(A, B).

goal2(X, Y) :- a(X, Y), b(X, Y).

a(C, C).
```

The purpose of data flow analysis [11, 13, 68] for FPPM is to obtain type and aliasing information for the variables in predicate arguments to simplify handling of data dependencies between parallel tasks†. We do not need a complete data flow information because Prolog's single

† Type information is also useful for sequential code optimization.

assignment semantics‡ simplifies the task of handling data dependencies as explained below.

If flow analysis determines that a predicate will be called with a variable in the arguments bound to a particular non-variable type, then the task that executes the predicate can wait until the variable is bound to a term of that type; we do not need the data flow graph because we do not need to know which task performed the binding.

Data dependency handling is more complicated if flow analysis is unable to obtain useful information about a set of variables in a predicate. In such cases we must assume that the variables could be unbound and aliased in any possible combination. We refer to such variables as *potentially shared unbound variables*. Parallel tasks must synchronize before binding potentially shared unbound variables. We illustrate the reasons for this with the following example.

```
 ?- goal3(X).

goal3(A)  :-  a(A),  b(A).

a(1).
a(2).

b(2).
b(1).
```

In the example above, the first solution produced by conventional sequential Prolog is `X = 1` and the second solution is `X = 2`. Next, suppose that `a/1` and `b/1` are executed by parallel tasks and that flow analysis did not give any useful information about the shared variable `A`. If there were no synchronization between the two tasks there could be two problems:

(1) The task for `a/1` attempts to bind `A` to `1` while the task for `b/1` attempts to bind `A` to `2` simultaneously.

(2) The task for `b/1` binds `A` to `2` first resulting in a different order for solutions than conventional Prolog semantics.

We handle potentially shared unbound variables in FPPM with a simple approach: If a task encounters such a variable it waits until either

‡ Logical variables can only be bound once during forward execution.

(1) the variable is bound to a non-variable term by another task, in which case the variable is no longer unbound, or

(2) all previous goals have completed, in which case the task may safely bind the variable. FPPM provides architectural support to simplify checking for this condition. We describe this support in chapter 4.

The flow analysis used in this dissertation is based on abstract interpretation and was implemented by Van Roy [69] as part of his Prolog compiler.

3.2. Program Partitioning and Task Scheduling

Many researchers have tried to exploit AND- and OR-parallelism in Prolog. There are several variations (reviewed in chapter 2), but the basic idea is that parallel tasks are created for the goals in a clause (to exploit AND-parallelism) and for the clauses in a predicate (to exploit OR-parallelism). However, because of the large overhead incurred by the complex memory management and task management schemes required to implement these systems, they rarely perform better than the most efficient sequential Prolog systems. The program partitioning scheme used in this dissertation is more selective. Parallel tasks are created only if they can be executed without large memory management and task management overhead.

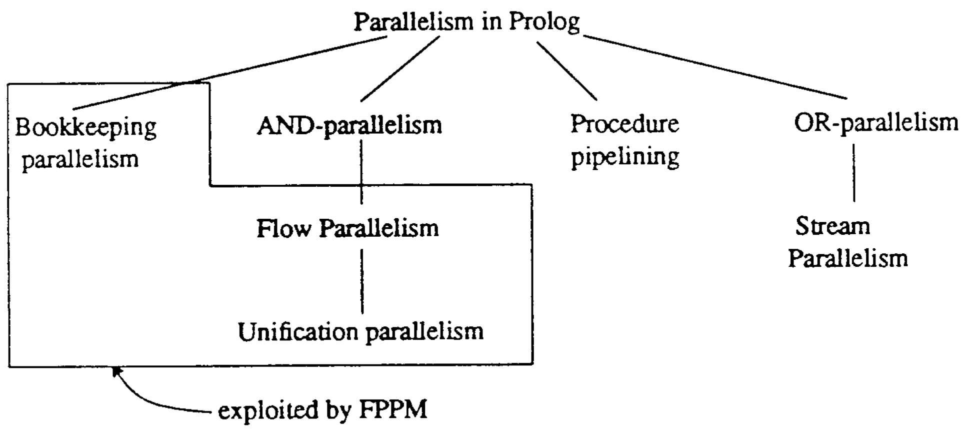


Figure 3.1: Taxonomy of Parallelism in Prolog

In section 3.2.1, we identify two major causes of overhead in other parallel execution schemes: side-effects† and choicepoints. We define *flow goals* as those that neither execute side-effects nor create choicepoints. Our partitioning algorithm creates parallel tasks only for flow goals. In section 3.2.2 we show that unification parallelism is a special case of flow parallelism. It follows that unification parallelism can be exploited with the same techniques as those used to exploit flow parallelism. Figure 3.1 illustrates the taxonomy of parallelism in Prolog. FPPM exploits flow parallelism (which includes unification parallelism). FPPM also exploits some bookkeeping parallelism by executing some bookkeeping tasks, such as environment allocation and deallocation, in parallel. However, other bookkeeping tasks, such as choicepoint creation and backtracking, are executed sequentially.

Context switching of tasks is another cause of overhead in parallel systems. Unless special care is taken in the program partitioning and scheduling methods, it may be necessary for a processor to suspend a task and switch contexts in order to avoid a deadlock. We explain the requirements for deadlock-free execution in section 3.2.3. We choose our partitioning and scheduling algorithms in such a way that deadlock can be avoided without resorting to context switching. An alternative scheme for program partitioning and task scheduling that also avoids context switching is described in section 3.3. This alternative scheme exposes more parallelism, but also incurs greater scheduling overhead. We compare the performance of the two schemes in chapter 6.

3.2.1. Problems with Side-Effects and Choicepoints

If sequential semantics of Prolog are to be maintained, then the side-effects that are directly visible to the programmer, such as input/output, must execute in the same order as in sequential Prolog. Similar restrictions on the order of side-effects that are not directly visible to the programmer, such as asserts and retracts, are also sometimes necessary to preserve sequential semantics. If parallel tasks can cause these side-effects then synchronization between tasks is required to enforce this order. This synchronization results in relatively complex parallel execution with higher overhead.

† Side effects are actions whose effects would remain should backtracking occur. Examples of side-effects are input/output, asserts and retracts.

For example, one solution [6] is to pass a token from task to task in the order in which the tasks would have executed if execution were sequential. A task may cause a side-effect only if it possesses the token and the token is passed to the next task only when the task has completed all its side-effects. The problem with this scheme is that it imposes the overhead of passing the token to all tasks although only a few have side-effects.

Allowing parallel tasks to create choicepoints creates the following problems:

(1) If a choicepoint remains after a task has completed execution then the stacks (environment and choicepoint) for that task cannot be deallocated and used for execution of another task because the data on the stack may be required on backtracking. Consequently, new stacks must be allocated for each task.

(2) Prolog's depth-first, left-to-right search imposes an order on choicepoints. The same order must be enforced for choicepoints created by the parallel tasks. This creates dependencies among the parallel tasks.

(3) In sequential Prolog the heap is usually allocated as a stack [71]. This has two advantages. First, space is recovered on failure, thus reducing the amount of garbage collection required. Second, variables above the heap pointer in the most recent choicepoint need not be trailed. Both these advantages can be exploited by parallel tasks if they do not create choicepoints since a parallel task can have space interleaved with that of other parallel tasks on the heap (although structures must still be contiguous). This is possible because all the tasks fail if any one of them fails (since there are no choicepoints between them). Therefore, all their heap space is restored on failure.

(4) Similarly, the trail stack space can also be recovered on backtracking if parallel tasks do not create choicepoints. In addition, since the trail entries for a task may not be contiguous, if the tasks create choicepoints they must be explicitly linked so that the variables bound by one task can be unbound if that task should fail. If the parallel tasks have separate stacks (both heap and trail) for each task then the size of the choicepoint increases since the stack pointers of all the tasks must be saved in a choicepoint. This increases the overhead of choicepoint

creation and backtracking.

(5) If backtracking is allowed within a parallel task, then environment frames cannot always be deallocated after returning from a procedure; the frame may be required if the procedure created a choicepoint. Thus, environment stack management is also complicated by backtracking.

3.2.2. Unification Parallelism and Flow Parallelism

Since they cause the problems described above, we disallow side-effects and choicepoints in parallel tasks. In our partitioning scheme, a goal is a candidate for execution by a parallel task only if it satisfies the following requirements:

(1) It can not create a choicepoint.

(2) It can not cause side-effects.

These conditions are satisfied for many predicates, in particular, for time-consuming inner loop predicates.

We call goals that satisfy conditions (1) and (2) above *flow goals*†. Flow parallelism is exploited when two or more flow goals are executed in parallel. Clearly, flow parallelism is a special case of AND-parallelism. It is easy to show that unification parallelism is a special case of flow parallelism and can be exploited with the same techniques. We illustrate this with the following example:

```
reverse([],[]).
reverse([X|L1],L):- reverse(L1,L2), concat(L2,[X],L).

concat([],X,X).
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3).
```

The program is easily transformed into the following equivalent program in which the head unification and the creation of complex goal arguments are written as new goals that perform expli-

† Goals that do not create choicepoints never undo their bindings (unless they fail entirely). If such a goal binds a variable, then information "flows" from the goal to other goals that share the variable.

cit unifications.

```
reverse(H1,H2):- H1=[], H2=[].
reverse(H1,H2):- H1=[X|L1], A1=[X], reverse(L1,L2), concat(L2,A1,H2).

concat(H1,H2,H3):- H1=[], H2=H3.
concat(H1,H2,H3):- H1=[X|L1], H3=[X|L3], concat(L1,H2,L3).
```

The purpose of rewriting the program in this way is to illustrate the following:

(1) Head unification and creation of goal arguments are similar since both can be treated as goals (called unification goals) that perform explicit unifications. For example, the goal `H1 = [X|L1]` in the second clause of `reverse/2` represents head unification while the goal `A1 = [X]` represents argument creation for the `concat/3` goal in the same clause.

(2) Parallel unification is exploited when two or more such goals are executed by parallel tasks. An example is the parallel execution of the goals `H1 = [X|L1]` and `H3 = [X|L3]` in the second clause of `concat/3`. Since unification goals do not create choicepoints nor execute side-effects, they are also flow goals.

3.2.3. Preventing Deadlock Without Context Switching

Context switching of tasks by processors also contributes to overhead in parallel execution. In order to eliminate this overhead, we disallow context switching in FPPM. However, unless program partitioning and scheduling algorithms are chosen appropriately, context switching may be necessary to avoid deadlock. In this section we describe constraints on our program partitioning and task scheduling methods that guarantee deadlock-free execution without the need for context switching.

Deadlock can arise if and only if all four of the following conditions hold simultaneously [52]:

(1) **Mutual exclusion.** A resource cannot be used by more than one task at the same time.

(2) **Hold and wait.** At least one task needs simultaneous access to two or more resources and it will not relinquish the resources that it has already been granted while it waits for the other resources.

(3) **No preemption**. Once granted, resources cannot be taken away.

(4) **Circular wait**. There must exist a set $\{t_0, t_1, \ldots, t_n\}$ of waiting tasks such that t_0 is waiting for a resource that is currently held by t_1, t_1 is waiting for a resource that is currently held by t_2, ..., t_{n-1} is waiting for a resource that is currently held by t_0, with $n > 0$.

Deadlock will not occur if we can ensure that at least one of the conditions above do not hold.

In the context of FPPM, there are two types of resources: (1) processors for tasks to run on and (2) input data for a task that is produced by another task. If context switching is disallowed, then the condition (1), mutual exclusion, holds for the processors. Condition (3), no preemption, also clearly holds for both types of resources if context switching is disallowed. In order to avoid deadlock without context switching, we must ensure that one of the remaining two conditions (2 and 4) do not hold. Condition (2), hold and wait, holds in the following cases: (a) a task is executing on a processor and is waiting for data from another task, or (b) a task that produces data needed by another task is waiting for a processor. Condition (4), circular wait, is satisfied if the tasks executing on the processors are waiting for data produced by tasks that are waiting for processors.

Consider a representation of a parallel program by a directed graph in which the tasks are the nodes and the data dependencies between the tasks are the directed edges. If the graph contains a cycle and the number of nodes in the cycle is greater than the number of processors, then all four conditions for deadlock hold if context switching is not allowed. In order to avoid this situation, we ensure that the program is partitioned such that the graph is acyclic. This requirement is met by a partition in which the tasks are Prolog goals since each goal depends only on goals that appear earlier in the sequential (depth-first, left-to-right) order of Prolog's execution.

A program partition with an acyclic dependence graph is not sufficient to avoid deadlock without context switching, as the following example illustrates. Consider a program consisting of three tasks, A, B, and C such that B and C depend on A for data. Suppose there are only two available processors and that tasks B and C have been scheduled to run on them while task A is waiting for a free processor. Clearly, the program is deadlocked if neither B nor C will suspend so that A can be executed. To ensure deadlock-free execution without context switching, the task scheduling algorithm must also be chosen appropriately.

We have already mentioned that our program partition can be represented by a directed acyclic graph (dag). We define a partial order, <, on the graph such that $A < B$ if and only if there is path from A to B in the graph (i.e., B depends on A). We choose a scheduling algorithm that ensures that all tasks, A, such that $A < B$, have been scheduled on processors before task B is scheduled. It is easy to see that deadlock will not occur with such a scheduling algorithm because condition (4), circular wait, does not hold. None of the tasks that currently have processors are waiting for data from a task that does not have a processor.

We simplify the implementation of such a scheduling algorithm by placing a restriction on the program partition (we will explain the reasons for such a restriction shortly). The parallel program consists of a single main program task and a number of parallel tasks. Only the main program is allowed to create parallel tasks. Each parallel task executes one or more goals for the main program and the order in which the tasks are created is the same as the sequential order of execution of these goals. Since parallel tasks are created only for flow goals, choicepoints and side-effects must be executed by the main program task. Figure 3.2 illustrates parallel execution in FPPM. With this partition, task scheduling is simple: tasks are scheduled in the order in which they are created by the main program.

This scheduling method (i.e., scheduling tasks in the order in which they are created) will result in deadlock-free execution only if parallel tasks are not allowed to create other parallel tasks. As the following example illustrates, deadlock can occur if parallel tasks are allowed to create other parallel tasks and context switching is disallowed.

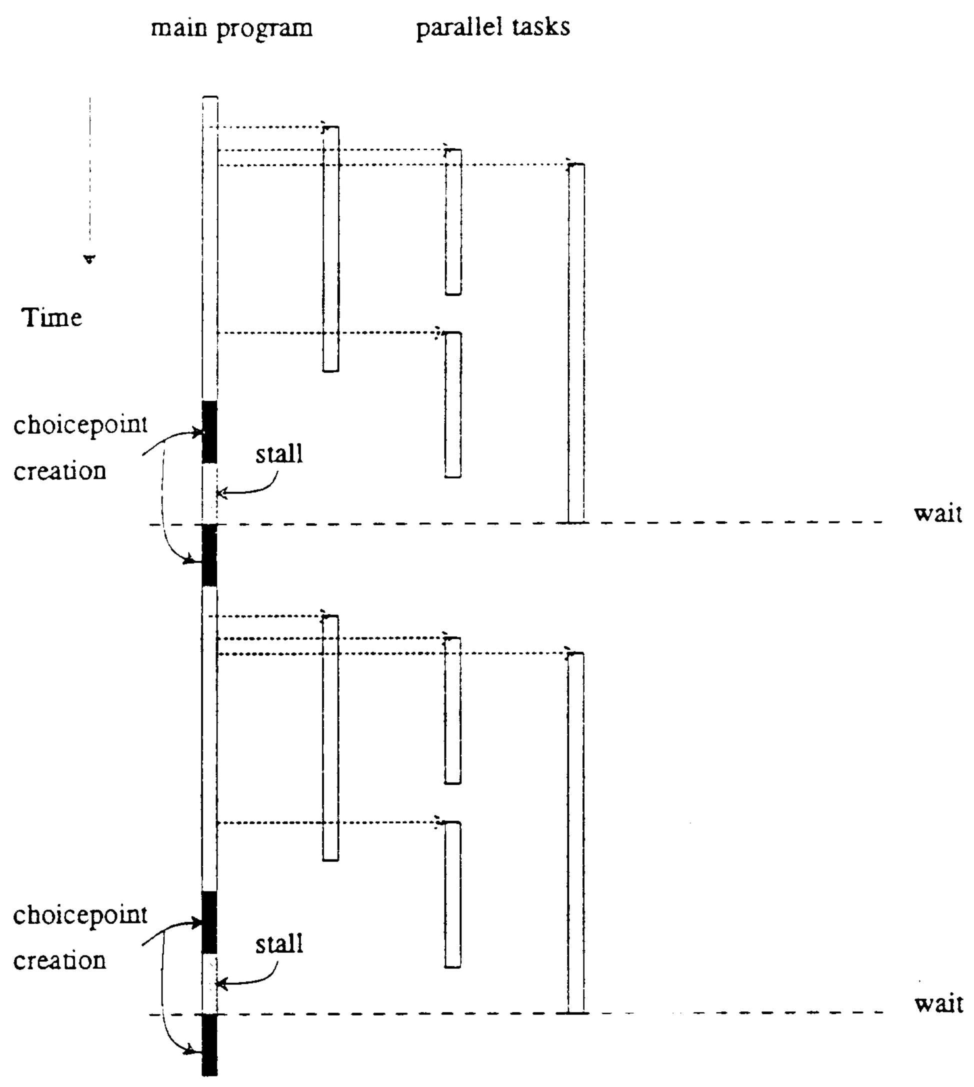


Figure 3.2: Task execution and choicepoint creation in FPPM. Only the main program can create parallel tasks. Choicepoints and side-effects are executed only by the main program. Choicepoint creation is partially overlapped with execution of other tasks, but all parallel tasks must terminate before the choicepoint creation can complete.

```
p:- a(X), b(X).

a(X):- a1(X), a2(X).

b(X):- b1(X), b2(X).
```

In the program above, assume that `a1/1` < `b1/1`. Suppose parallel tasks are created for execution of the goals `a/1` and `b/1` in that order. Now, if the tasks for `a/1` and `b/1` are allowed to create parallel tasks for their goals, it is possible that `b/1` creates a task for `b1/1` before `a/1` creates a task for `a1/1`. The scheduler would then schedule the task for `b1/1` before `a1/1`. If there is no other free processor for `a1/1` then a deadlock occurs. Therefore, if parallel tasks are allowed to create other parallel tasks then either context switching should be allowed or a more sophisticated scheduling algorithm should be used. Both options result in greater overhead. We describe one scheme that does not need context switching and allows parallel tasks to create other parallel tasks in section 3.3. The scheme is evaluated in chapter 6.

Our partitioning and scheduling methods have the following advantages:

- No task suspension and context switching are required.
- Scheduling can be implemented using a single queue of tasks. Tasks may be added to the queue without synchronization by the processor that executes the main program task since there is only one processor adding tasks to the queue; synchronization is only required for the processors that take tasks off the head of the queue.
- The scheduling overhead can be significantly reduced by architectural support described in chapter 4.

The main disadvantages of such a scheme stem from the fact that only one task is allowed to create other parallel tasks:

- The scheme cannot exploit a large number of processors because the performance is limited by the fact that parallel tasks can be created by only one processor.
- Parallelism cannot be exploited effectively for for some programs. In particular, the amount of parallelism that can be exploited in non-linearly recursive Prolog programs is limited (we explain the reasons for this later in this chapter).

In spite of these disadvantages, we show that our partitioning and task scheduling method can achieve good speedups over a variety of programs for multiprocessors with a small number of processors (4 to 8). Our scheme is probably not the right choice for multiprocessors with a large number of processors.

3.2.4. Heuristics for Program Partitioning

Determining the granularity of a task (i.e., its execution time) is an extremely hard problem. In general, it is not solvable since it is equivalent to the halting problem. Therefore, the best that one can hope for is an estimate. Sarkar [55] uses profile information from sequential execution of the program to obtain estimates of task sizes. In cases where the task size depends on the value of input data, some researchers, for example Fagin [29] and Shu, *et al* [57], have used explicit tests on input data to decide whether to create a parallel process for a goal or not. Although they currently require the programmer to explicitly provide these tests, they claim that they expect future compilers to automatically generate the tests. Automatic generation of such tests is extremely hard and requires a sophisticated program analysis that, to the best of our knowledge, is not yet in sight. Most parallel systems are very dependent on such tests because parallel task creation has a high overhead that can easily result in slower execution rather than a speedup of the program if the test is not selected judiciously.

Rather than rely on profile information, or generate tests to determine at run time whether or not to create a parallel task, we rely on a static analysis of the program's structure to determine which goals will execute as parallel tasks. This analysis is much simpler than that required for generating run time tests on input data because it does not consider how the input data influences the execution time. Rather, it assumes that a loop (recursion) does sufficient work to execute as a separate (parallel) task. Given this knowledge of recursions in the program we then use heuristics to select those flow goals for which parallel tasks are created. We consider heuristics for two cases: linearly recursive predicates and non-linearly recursive predicates.

3.2.4.1. Linearly Recursive Predicates

Prolog programs use recursion to implement iteration. We use the the structure of recursion to partition the program into parallel tasks. The simplest recursive clause has only one recursive call. Such clauses are called *linearly recursive*. Predicates whose clauses are either non-recursive or linearly recursive are called linearly recursive predicates. In this section we discuss only linearly recursive predicates, leaving the non-linearly recursive predicates for the next section. An example of a linearly recursive clause is given below.

```
concat([X|L1], L2, [X|L3]):- concat(L1, L2, L3).
```

It is clearly not useful to create a parallel task for the only goal of the recursive clause since this goal would then execute sequentially as a task. However, if the clause also has some computation before or after the recursive call, then that computation can be executed as a parallel task for each iteration. In fact, the clause above does have some computation before the recursive call. This computation becomes apparent if we rewrite the clause as follows:

```
concat(H1, L2, H2):- H1=[X|L1], H2=[X|L3], concat(L1, L2, L3).
```

We now have two unification goals for every iteration that can be executed as parallel tasks. To simplify the discussion, we assume that both the unification goals for each iteration are executed by a single task, although the two unifications could also be executed as separate parallel tasks. Parallel execution of the clause is illustrated in figure 3.3. Goals that appear after the recursive goal may also be executed as parallel tasks. An example of clause that has goals before and after the recursive goal is

```
reverse(H1,H2):- H1=[X|L1], A1=[X], reverse(L1,L2), concat(L2,A1,H2).
```

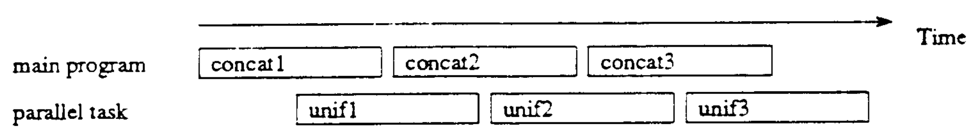


Figure 3.3: Parallel execution for computations before linearly recursive goals.

Again, for simplicity we assume that both the goals before the recursive goal are executed by a single task. Parallel execution of the clause is illustrated in figure 3.4. In figure 3.4 the boxes marked "rev 1" through "rev 3" represent execution of the clause for three iterations respectively. The box marked "rev term" represents the execution of the base case (which terminates the recursion). The box marked "revt 3" through "revt 1" represent the execution of the three iterations after the recursive goals. Figure 3.4 also shows that it is possible that parallel tasks for multiple iterations

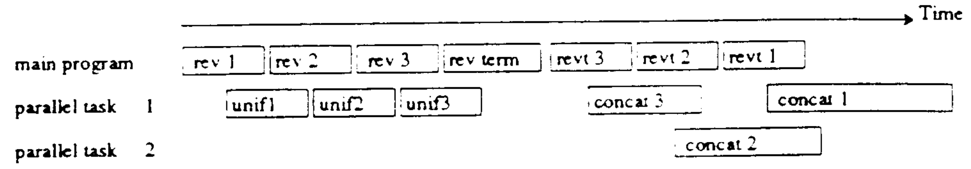


Figure 3.4: Parallel execution for computations before and after linearly recursive goals.

may execute simultaneously.

The examples above serve to illustrate some of the choices that the partitioning algorithm must make. In the recursive clause of the `reverse/2` predicate above we created a task for each `concat/3` goal in the clause. Since parallel tasks in FPPM are not allowed to create other parallel tasks, no parallelism can be exploited within the `concat/3` goals. However, we can also exploit parallelism of a finer grain in the `concat/3` goals since the recursive clause of `concat/3` represents a nested iteration. For which goals should the compiler generate parallel tasks: the unification goals of the `concat/3` clause or the `concat/3` goals of the `reverse/2` clause? We do not hope for optimal solutions to these choices. Instead we rely on a heuristic, which, as we shall see in chapter 6, works quite well.

Partitioning Heuristic 1: Favor Outer Loop

If there are nested linearly recursive predicates, then create a parallel task for each innermost recursive loop.

For example, in the program below, the predicate `inner` is the innermost loop and the predicate *outer* is the outer loop. The clauses `before` and `after` are goals that do not contain loops. Parallel tasks for the goal `inner` are created during each execution of the recursive clause for `outer`. If there are more than one goals that represent inner loops, then a parallel task is created for each such goal.

```
outer(..) :- before, inner(..), outer(..), after.
outer(..).

inner(..) :- inner(..).
inner(..).
```

Partitioning Heuristic 2: Minimum Size

Create parallel tasks for goals that contain no loops (recursions) if they contain enough work.

In this dissertation we do not discuss how one might determine whether or not a goal with no loops contains enough work; the heuristic could use criteria such as the minimum length of the instruction sequence that the goal executes.

For example, in addition to creating a parallel task for the `inner` goal in the recursive clause of `outer`, parallel tasks may be created for the goals `before` and `after` if they contain computations that are large enough to justify creation of a parallel task.

3.2.4.2. Non-Linearly Recursive Predicates

A non-linearly recursive clause is one that has more than one recursive goal. A Non-linearly recursive predicate is one that has a non-linearly recursive clause. In most parallel systems a parallel task is created for each recursive call. Each parallel task, in turn, creates parallel tasks for each of its recursive calls. However, our model of execution does not allow parallel tasks to create other parallel tasks. This limits the amount of parallelism that we can exploit in non-linearly recursive predicates. In order to exploit more parallelism in non-linearly recursive predicates we would have to allow parallel tasks to create parallel tasks. But if we allowed parallel tasks to create other parallel tasks then, in order to avoid deadlock, we would have to either (1) allow task suspension and context switching or (2) use different program partitioning and task scheduling methods. Both options are likely to increase complexity and parallel execution overhead. In keeping with the goals of this dissertation, we choose to retain our low overhead model of execution in spite of the limited parallelism that it can exploit.

Partitioning Heuristic 3: Last recursive goal

Treat the non-linear recursion as a nested linear recursion by considering only the last recursive goal as a recursive call; the other recursive calls are treated as inner loops and executed as parallel tasks that cannot create other parallel tasks.

Non-linearly recursive predicates are often used to specify divide-and-conquer algorithms. An example of such a predicate is the quicksort (`qsort/2`) predicate shown below.

```
qsort([],[]).
qsort([X|L],S):-
    divide(X,L,L1,L2),
    qsort(L1,L1s),
    qsort(L2,L2s),
    concat(L1,[X|L2],S).
```

Heuristic 3 creates parallel tasks for the `divide/4`, the first `qsort/3` and the `concat/3` predicates; the second `qsort/3` goal is executed by the main program and forms the outer loop. This requires two versions of the compiled code for the `qsort/3` predicate: one for execution as the main program, and the other for execution as a parallel task. The parallel task for `concat/3` is created only after returning from the recursive call to the the second `qsort/3` goal. The execution of this program is illustrated in figure 3.5.

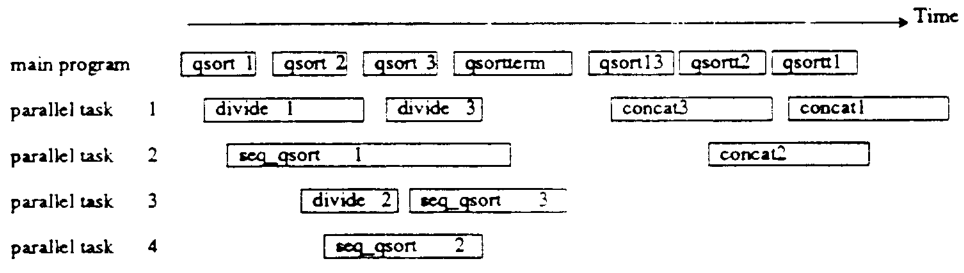


Figure 3.5: Parallel execution for non-linearly recursive predicates

Figure 3.6 illustrates the main problem with the partitioning scheme: the execution time could be dominated by a few large tasks. One disadvantage of an alternate heuristic, in which the main program follows all the non-linearly recursive calls instead of just the last one, is apparent from figure 3.7: there are many more small tasks, resulting in higher overhead. A second disadvantage of

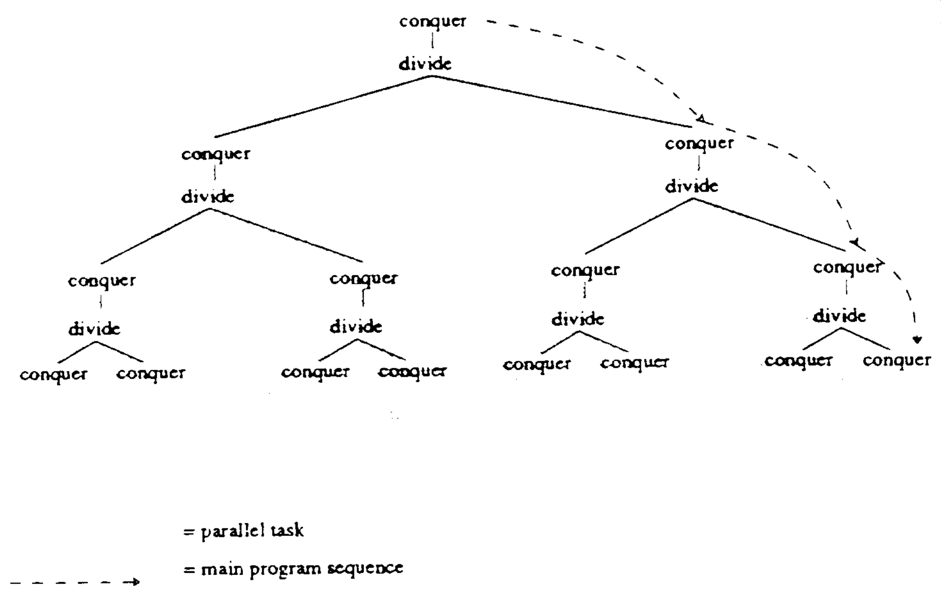


Figure 3.6: Partitioning of divide-and-conquer problem in FPPM

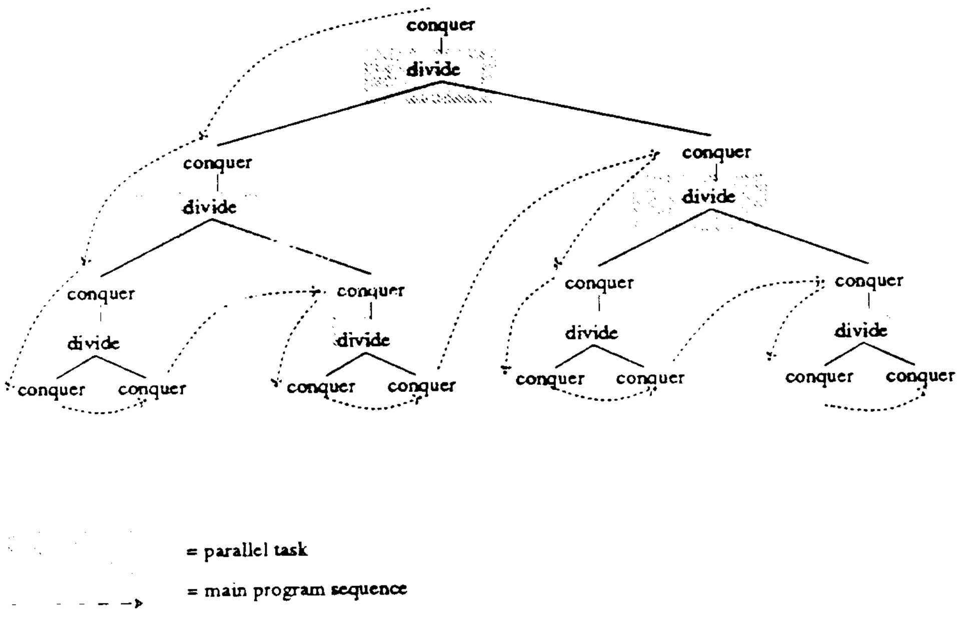


Figure 3.7: An alternative partition of the divide-and-conquer problem

this alternative heuristic is more subtle and has to do with how often the parallel tasks are busy-waiting for data from other tasks. Each task for the `divide` goal divides the data for the original problem (the input list in the case of quicksort) into two sets of data. Each of these sets of data are, in turn, divided into smaller sets by subsequent `divide` tasks. Consequently, each of these tasks have progressively smaller sets of data to work on. However, in the alternative partitioning scheme, `divide` tasks are created first for only one of the sets along a depth-first path as shown in figure 3.7; the other sets are not processed further until later, when the main processor returns from the first recursive call and executes the second. In FPPM, a `divide` task does not wait until its input is complete: it operates on whatever data the previous task has placed in its input and busy-waits for further data. Since each task operates on only a part of its predecessor's output, each subsequent task must busy-wait more than its predecessor. Consequently, average processor utilization can be poor.

3.2.4.3. Mutually Recursive Predicates

With mutually recursive predicates any one predicate in the recursion could be chosen as the predicate that executes in the main program with the others executed as parallel tasks. This requires the predicate that executes on the main processor to be compiled in two versions since it may also execute sequentially within a parallel task. Two compiled versions of predicates may be required in other instances as well if the predicate is executed in the main program as well as within a parallel task.

3.3. An Alternative Scheme for Partitioning and Scheduling

As mentioned earlier, a major disadvantage of FPPM's methods for partitioning and scheduling is that the parallelism that is exploited in non-linearly recursive predicates is limited because parallel tasks can not create other parallel tasks. In this section, we describe an alternative scheme in which parallel tasks may create other parallel tasks. As in the earlier scheme, we ensure that context switching is not necessary to avoid deadlock. Although the scheme exploits more parallelism in non-linearly recursive predicates, it incurs greater scheduling overhead. We evaluate this scheme in chapter 6.

In this scheme, we allow parallel tasks to create other parallel tasks. However, in order to ensure that context switching is not necessary to avoid deadlock, a task creates a parallel task to execute a flow goal only if a processor is available to execute the task; if no processor is available, then goal is executed (as in conventional sequential Prolog execution) by the same task. We implement this method by having the processors queue for tasks, instead of queuing the tasks for processors. Deadlock will not occur because no task ever waits for a processor.

The additional overhead in this scheme arises due to the following reasons:

(1) For each goal that can be executed as a parallel task, the code must first check to see if a processor is available. This check is overhead if no processor is available. If a processor is available, then it is removed from the queue and the task is dispatched to it.

(2) No synchronization is needed to create tasks if a single processor creates the tasks; synchronization is only required when processors take tasks out of the task queue. If processors queue for tasks and parallel tasks are allowed to create other parallel tasks, synchronization is required for processors to enter the queue and for tasks that take processors from the queue.

(3) The processors in the queue are idle. If tasks are queued instead of processors, then fewer processors are idle because the task queue acts as a buffer for waiting tasks.

(4) If more than one processor can create parallel tasks, then the architectural support for low overhead task creation in FPPM cannot be fully exploited.

3.4. Chapter Summary

In this chapter, we have described our approach to (1) handling data dependencies, (2) program partitioning and (3) task scheduling. In each case, we use low overhead schemes, even though more complex methods can exploit more parallelism at the expense of greater overhead. We explained how side-effects and choicepoints in parallel tasks contribute to overhead. Consequently, in the first scheme we restrict parallel execution to flow goals. We also restrict the program partition so that only a single main program is allowed to create other parallel tasks. This enables us to use a low overhead task scheduling method that schedules tasks in the order that they are created. With such a scheme, task suspension and context switching are not required to avoid deadlock. Tasks may busy-wait for data from other tasks, thus eliminating the overhead due to context switching. We also described heuristics for partitioning programs into parallel tasks at compile time.

If parallel tasks are not allowed to create other parallel tasks, the amount of parallelism that can be exploited in non-linearly recursive predicates is limited. Therefore, we described an alternative scheme that allows the parallel tasks to create other parallel tasks. However, this alternative scheme incurs greater scheduling overhead.

4. FPPM: An Architecture to Exploit Flow Parallelism

In this chapter, we describe a new architecture, the Flow Parallel Prolog Machine (FPPM), that exploits flow parallelism in Prolog. FPPM has special architectural features to support

- sequential Prolog execution,
- data dependency handling,
- low overhead task creation and termination.

Figure 4.1 is an overview of the FPPM architecture. FPPM consists of a Main Processor that executes the main program and one or more attached processors, called Slave Processors, that execute the parallel tasks created by the main program. All the processors have access to shared memory. Processors also share a two types of registers: global registers and write-once registers. The global shared registers allow tasks to share stack pointers efficiently. The write-once registers allow tasks to pass arguments to other tasks with very low overhead.

Section 4.1 describes the data representation in FPPM. Section 4.2 describes the FPPM registers. Although most instructions are common to both the Main Processor and the Slave Processors, each type of processor has some specialized instructions. Section 4.3 describes the instruction set common to both the Main Processor and the Slave Processors. Section 4.4 describes the instructions specific to the Main Processor and section 4.5 describes the instructions specific to the Slave Processors. Section 4.6 describes how programs are compiled for FPPM.

4.1. Data Representation

Prolog has dynamic typing, which means that variables can be bound to values of different types at run time. Therefore, it is necessary to encode the type of each data item along with its value. We store the type and value of a data items in the same word so that only one word needs to be read for each data item. The type is encoded in a type tag field in the word.

Data words in FPPM are 32 bits wide and consist of a 4-bit type tag and 28-bit value field. Two type tag values are defined by the architecture: tag 0 represents an unbound variable and tag 1 represents a reference or bound variable. All other tags are defined by the software. It is useful to

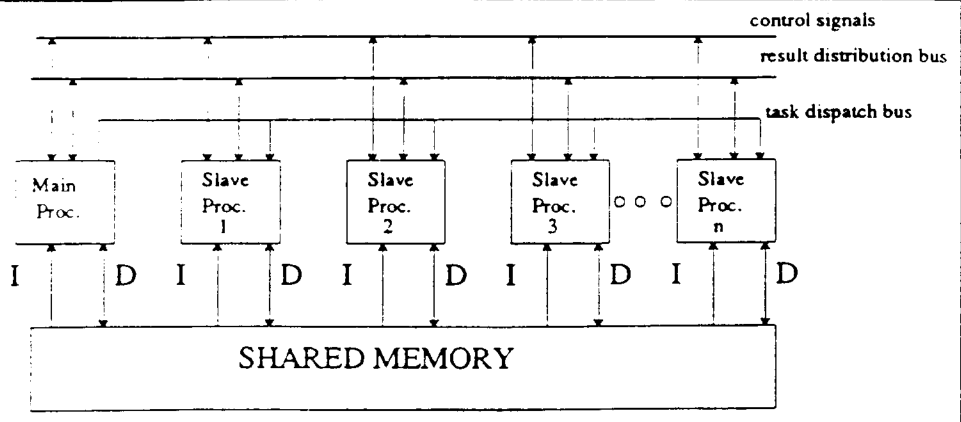


Figure 4.1: Overview of FPPM

specify tags for unbound and bound variables in order to support dereferencing of variables efficiently in the architecture. The 28-bit value field of the data word can contain word addresses or values. Table 4.1 lists some type tags used in this dissertation for lists, structures, integers and atoms.

tag (hex)	type
0	unbound variable
1	bound variable (reference)
2	list
3	structure
4	integer
5	atom
6 - E	(currently undefined)
F	nil (special constant)

Table 4.1: Data Type Tags in FPPM

Simple data types are represented by a single data word. The type is indicated by the tag field. The value field can either hold the value of the datum (as in an integer of 28 bits or less) or a pointer to the value as an atom where the value field contains a pointer to the symbol table entry for the name of the atom. Structure data types of arity *n* are represented by a single data word containing a structure tag and a pointer to the first word of a contiguous chunk of *n* + 1 memory locations.

The first word of the chunk is an atom representing the functor and arity of the structure. Its value field is a pointer into the symbol table entry for the functor. The arity of the structure is stored in the symbol table. The remaining *n* words contain the arguments of the structure. Since lists are structures of arity 2 and fixed (but unspecified) name, the functor field is omitted in list representations. Lists occur often enough in practice that this optimization is quite useful and is used in all Prolog systems that we know of. A special tag is provided for lists. List data types are represented by a word containing a list tag and a pointer to the first word in a chunk of 2 contiguous memory locations. The first word contains the first element of the list (the *car*) and the second is a representation of the rest of the list (the *cdr*). Figure 4.2 illustrates representations for data structures in the FPPM.

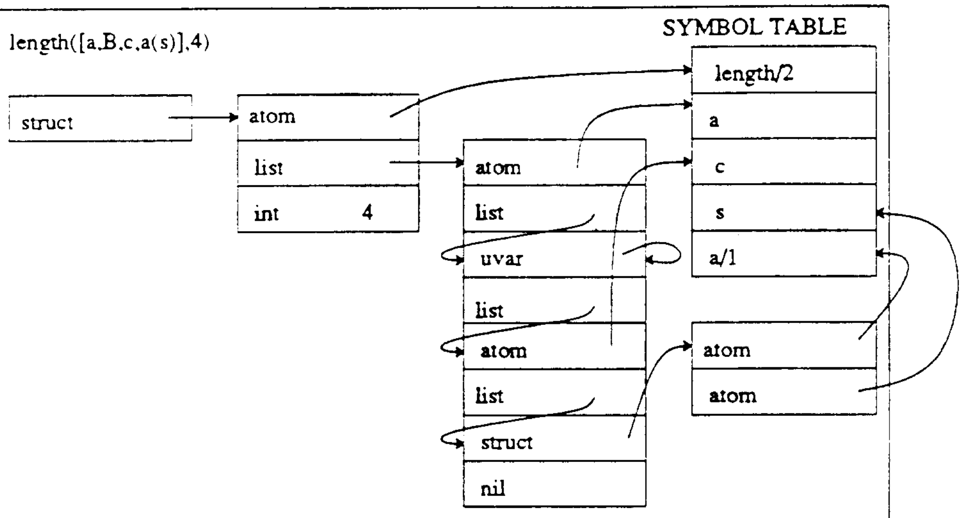


Figure 4.2: Representation of "length([a,B,c,a(s)],4)".

Dobry *et al* [22] employ a more concise representation for lists using *cdr-coding*, in which a special cdr-bit is used to indicate that the word is the cdr of a list. Cdr-coding has two disadvantages: (1) Unification algorithms for cdr-coded lists are more complicated [25]; (2) the cdr-bit uses an extra bit in each word. Touati and Despain [66] have shown that the space saved by the more compact list representations due to cdr-coding is rather small because most lists are constructed during program execution by unifying a list with an unbound variable in the cdr of another list. Lists constructed in this way do not use cdr-coding.

4.2. The Registers

Each processor in FPPM (i.e., the Main Processor and the Slave Processor) has access to three register files: (1) the local register file, (2) the global register file and (3) the write-once register file. The local register file consists of 16 registers. As the name implies, the local registers of one processor cannot be accessed by another processor. An instruction never stalls when reading or writing a local register.

Global Registers

The global register file consists of 16 shared registers. A new value written to a global register by any processor will be seen by all processors at the same time. We describe the implementation of the registers in more detail in chapter 5, but it is necessary to understand the general method operation of the global registers in order to understand the FPPM architecture. We describe the implementation very briefly below. The global registers are implemented by shadow copies of the register file in each processor. Global registers are only updated over a shared broadcast bus (called the *distribution bus*). Processors arbitrate for the use of the bus. Therefore, unlike an instruction that writes to a local register, an instruction that writes to a global register will stall if it is not granted the distribution bus in the first round of arbitration. An instruction never stalls when reading a global register. Another important difference between an instruction that writes to a local register and one that writes to a global register is that local register writes are bypassed†, whereas global register writes are not. We do not allow global register writes to be bypassed because the bypass paths are local to processors (i.e., the processor that writes to the global register can use the new value before all the others, violating the requirement that all updates be simultaneously visible to all processors). Since there are no bypass paths for global registers, a value written to a global register can only be read four instructions later.

Global registers are used for shared stack pointers, such as the heap pointer and the trail pointer. A processor that wants to allocate heap space must (1) obtain exclusive access to the heap

† A bypass path allows the data that is written to a register to be used by subsequent instructions as soon as it is available in the data path.

pointer (the architecture provides a lock for this), (2) read the shared heap pointer and increment it by the amount of space required, and (3) release exclusive access to the heap pointer.

Write-Once Registers

The write-once register file consists of multiple sets of shared registers with valid bits associated with each register. Each register set contains 16 registers. The architecture specifies a minimum of 3 write-once register sets. The actual number of register sets may vary with the implementation. Like the global registers, write-once registers are implemented by shadow copies of the register file on each processor with updates occurring only over the shared distribution bus and are not bypassed. Updates to a write-once register have the same latency as global registers (i.e., four instructions). Unlike global registers, each write-once register has a valid bit that is set when the register is written, and instructions that read a write-once register will stall if the valid bit is not set, since this indicates that the data that the instruction intended to read is not yet in the register. The valid bit can be used to implement simple data flow control provided the valid bit of the register is reset after its value is no longer required and before the new value is either written or read. It is the responsibility of the compiler to ensure that a write-once register is written only once before its valid bit is reset.

Each instruction in a processor has access to registers in only two adjacent write-once register sets: an input set and an output set. The input set usually contains arguments to a procedure. The procedure usually writes only to the output set. However, an instruction may read or write a register from either set. The output set of one procedure is the input set of another. A special instruction (i.e., *new*) executed on the Main Processor makes the current output set the new input set and provides a new output set in which all the valid bits have been cleared. The instruction, *old*, reverts to the old input and output sets. Although the architecture assumes that there is an infinite supply of new register sets that can be obtained by executing *new*, an implementation can only have a small number (typically 4 or 8) of them. The implementation organizes the registers sets as a circular queue so that the set numbers wrap around and register sets may be re-used. The number of the input set for each instruction is referred to as the set number for that instruction. The *new* instruction stalls if there are any processors still executing instructions with the set number of the next output set. This ensures that the valid bits of registers in a set are not cleared until all instructions that could potentially read those registers are completed. The write-once registers can be used to pass arguments to tasks that have been created by the Main Processor.

code	mnemonic	type
0	IN	write-once input set
1	OUT	write-once output set
2	G	global
3	T	local

Table 4.2: Register Type Encodings in FPPM

A register operand is specified in a FPPM instruction using two fields: a 2-bit type field and a 4-bit register number field. Table 4.2 lists the various register types.

4.3. The Common Instruction Set

The common instruction set is recognized by both the Main Processor and the Slave Processors of FPPM. All instructions in FPPM occupy exactly 1 word (32 bits) The instructions are inspired by the intermediate language suggested by Van Roy in [67] for Prolog compilation.

The various instruction fields are specified in table 4.3. The lock and unlock bits are used to obtain and release a single lock provided by the hardware. The lock and unlock fields are provided to enable each instruction to lock and unlock a global lock for synchronization. The use of the lock and unlock are explained later. The special code field is used with **arith**, **brcond** and **brncond** instructions to indicate an arithmetic operation to perform or condition to branch on. The instruction formats are chosen such that the register source operands and branch address displacement can be quickly obtained.

The instruction set is best understood keeping in mind the basic 5-stage pipeline structure of the processors shown in figure 4.3. Registers are read in the decode stage (d-stage) and instructions that have invalid register operands stall in the d-stage. The d-stage also computes the branch addresses causing a branch delay of 1 cycle. "Squashed" versions of branch instructions are

range	field name
31	lock bit (lbit)
30	unlock bit (ulbit)
29:24	opcode
23:20	special code (scode)
23:20	immediate tag (itag)
19:12	offset (off)
23:12	immediate (imm)
19:6	label (lab)
23:0	long label (longlab)
17:16	register 2 type (reg2t)
15:12	register 2 number (reg2)
11:10	register 0 type (reg0t)
9:6	register 0 number (reg0)
5:4	register 1 type (reg1t)
3:0	register 1 number (reg1)

Table 4.3: Instruction Fields in FPPM

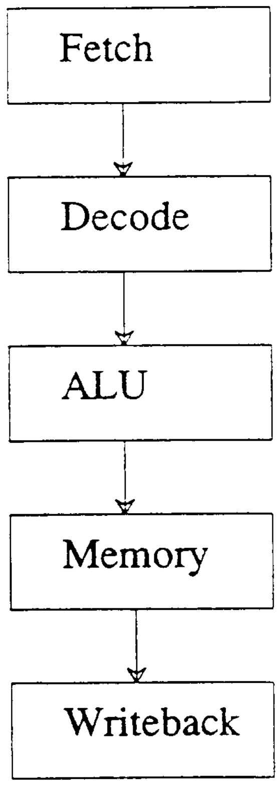


Figure 4.3: Basic Pipeline Structure for Processors

provided so that the instruction after the branch can be annulled if the branch is taken. Local register values are bypassed from the ALU stage (a-stage) and memory stage (m-stage) for instructions that use them as operands in the a-stage. Consequently local register values computed by instructions in the a-stage are available for the a-stage of the very next instruction and there is 1 load delay slot for local register values read from memory. Branch instructions that use local register values in the d-stage experience a 1-cycle delay for values computed by the a-stage and a 2-cycle load delay for values read from memory. No bypass paths exist for write-once and global registers.

If the lock bit in an instruction is set, then the processor executing the instruction requests a global lock before reading any of its operands (i.e., in the decode stage of the pipeline) and stalls unless the lock is granted. No other processor will be granted the lock until the processor releases it. The lock is released if the fail line for set number of the instruction that obtained the lock is set or if a subsequent instruction on the processor releases the lock. If the instruction has the unlock bit set then the lock is released after the destination registers writes have been initiated (i.e., in the last stage of the pipeline).

The instructions are divided into the four categories and listed in table 4.4 along with a description in register transfer notation.†

The processors can signal a failure of a unification to the other processors by executing the *fail* instruction (pulling a fail line high). There are separate fail lines for each set. The fail signal for a set causes (1) the Main Processor to abort and branch to an address in a fixed register (T,FAIL_ADDR), (2) all tasks of the failed set to be aborted, (3) the output write-once registers for the set to be cleared and (4) the locks granted for instructions of the set to be released.

The various arithmetic operation and branch condition encodings of the scode field are listed in table 4.5. The only field encoding that needs explanation is DRF. The DRF condition is true

† The tag and value parts of a word are indicated by ".t" and ".v" respectively. Concatenation is represented by "^". All the register specifiers are actually two fields, one specifying the register type and the other the register number. For write-once register types, the actual set number is computed by adding the type to the current set number for the instruction modulo the total number of register sets. The fetch stage program counter is denoted as "fpc" and the decode stage program counter (which points to the instruction *following* the one being decoded) is denoted as "dpc". There are "squashed" versions of every branch operation in which the instruction in the delay slot is annulled if the branch is taken. An "s" prefix in a branch instruction indicates a squashed version.

instruction	description
ALU	Instructions
nop	no operation
arith(scode,r1,r0,r2)	r1 <-r0.t^(r0.v scode r2.v)
arithoff(scode,r1,r0,off)	r1 <- r0.t^(r0.v scode off)
movetag(r1,r0,itag,off)	r1 <- itag^(r0.v + off)
Memory	Instructions
load(r1,r0,off)	r1 <- m(r0.v + off)
store(r1,r0,off)	m(r0.v + off) <- r1
push(r1,r0,off)	m(r1) <- r0.t^(r0.v + off); r1 <- r1.t^(r1.v + 1)
pushtag(r1,r0,itag,off)	m(r1) <- itag^(r0.v + off); r1 <- r1.t^(r1.v + 1)
Branch	Instructions
br(longlab)	fpc <- dpc + longlab
brind(r1)	fpc <- r1.v
bral(r1,lab)	fpc <- dpc + lab; r1 <- dpc + 1
brtageq(itag,r1,lab)	if r1.t == itag fpc <- dpc + lab
brtagneq(itag,r1,lab)	if r1.t != itag fpc <- dpc + lab
brcond(scode,r1,lab)	if scode(r1) fpc <- dpc + lab
brncond(scode,r1,lab)	if !scode(r1) fpc <- dpc + lab
brcmp(r1,r0,imm)	if r1 == r0 fpc <- dpc + imm
brncmp(r1,r0,imm)	if r1 != r0 fpc <- dpc + imm
Signaling	Instructions
fail	clear current set's tasks, interrupt Main Processor
waitexec	wait until previous tasks are done

Table 4.4: Common Instruction Set

mnemonic	operation
Arithmetic	Codes
ADD	addition
SUB	subtraction
NAND	nand
NOR	nor
SLL	shift left logical
SRL	shift right logical
MAX	maximum
Condition	Codes
GEZ	greater or equal to zero
EQZ	equal to zero
DRF	dereferenced
OTF	other fail

Table 4.5: Scode Field Encoding

when a word has been dereferenced. The condition is *not* satisfied if either (1) the word has a BVAR (bound variable) tag or (2) the word has a UVAR (unbound variable) tag and the instructions of some previous set are not complete. Condition (1) implies that the word is pointer to another word. Condition (2) implies that the word is an unbound variable, but since the instructions of the previous set are not complete there is a possibility that the variable will be bound by the instructions of the earlier sets.

4.4. Main Processor Specific Instructions

instruction	description
exec(longlab)	dispatch task starting at dpc + longlab
new	allocate new register set and clear its output set
old	go back to previous register set
clearfail	clear fail latch for current set
quit	halt

Table 4.6: Main Processor Specific Instructions

The Main Processor has special instructions to (1) dispatch tasks for execution on Slave Processors, (2) allocate new register sets and (3) handle failure of unifications. The Main Processor Specific Instructions are listed in table 4.6.

4.5. Slave Processor Specific Instructions

The Slave Processors have only one instruction in addition to the common instruction set: the **done** instruction. This instruction indicates that the current task is done and another task may be started. It is similar to a branch instruction in that the fetch stage program counter is loaded, and consequently there is a delay slot after the **done** instruction.

Instructions that belong to different tasks, but are currently in the pipeline of the same Slave Processor, are *decoupled* with respect to pipeline interlocks. In other words, stalls in the pipeline caused by instructions of the next task do not stall the instructions of the current task; the instructions of the current task are allowed to complete. This decoupling is very useful in avoiding deadlock situations due to instructions from different tasks being executed on the same processor.

For example, consider the situation in which the last instruction of a task writes to a write-once register, and the first instruction of the next task that executes on the same Slave Processor reads the same write-once register. When the first instruction of the second task is in the d-stage, the last instruction of the first task has not yet reached the w-stage. Since the write-once register has not yet been written, the instruction in the d-stage stalls. If the instructions are not decoupled, the stall in the d-stage also stalls the instructions in later stages of the pipe, including the one that writes the write-once register, causing a deadlock. The compiler would then have to ensure that deadlock situations do not arise by checking all possible combinations of tasks for deadlock.

4.6. Compiling for FPPM

In this section, we describe standard register usage and code sequences for common Prolog operations that could be generated by a compiler for FPPM. Note that the register usage, memory organization and code sequences in this section are only conventions used in this dissertation; of course, FPPM may be programmed in other ways as well.

4.6.1. Memory and Register Organization in FPPM

The basic data structures and memory organization of FPPM is derived from the WAM. In FPPM, memory is divided into a shared code area, a shared global stack (called a heap in WAM terminology), a shared trail stack area and separate local stacks for each processor. A single global register (G,HP) is used as the pointer to the top of the heap. Each processor allocates heap space in chunks and a local register (T,HP) is used to keep track of the top of the local chunk of heap space. The size of the remaining chunk of heap space is stored in another local register (T,HC). A single global register (G,TR) is used as a pointer to the top of the trail stack. The starting address of the local stack of each processor is stored in a local register (T,E). This register is never written on a Slave Processor. At the beginning of each task in the Slave Processors the (T,E) register is copied into the (T,TE) register which is then used as the pointer to the top of the local stack. The (T,E) register is never written because we want to be able to start a new task with an empty local stack even if the previous task that executed on the processor failed without restoring its stack pointer.

The local stack is used for environments on all processors and for choicepoints as well on the Main Processor. The Slave Processors do not create choicepoints. The Main Processor has one local register (T,B) that serves as the backtrack pointer (i.e. the pointer to the most recent choicepoint). The Main Processor aborts execution when any fail line is set and branches to the address stored in a fixed local register (T,FAIL_ADDR). On each processor, the temporary register (T,CP) is used as the continuation pointer (or linkage register).

4.6.2. Instruction Sequences for Operations Requiring Synchronization

We now describe how FPPM performs operations that require synchronization. One such operation is allocating a chunk of memory on top of the shared heap. The instruction sequence, represented as a macro by `getheap(Size)` is

```
getheap(Size) =

l_arithoff(ADD,G,HP,G,HP,Size) ! lock, (G,HP)new <- (G,HP)old + Size
u_arithoff(ADD,T,HP,G,HP,0) ! unlock, (T,HP) <- (G,HP)old
arithoff(ADD,T,HC,G,0,Size) ! (T,HC) <- Size
```

The trail operation also requires synchronization because the trail stack is shared. The trail operation is described below, where the register (T,R) contains the address of the variable:

```
l_push(G,TR,T,R,0) ! lock,(G,TR)<-(G,TR)+ 1, m[(G,TR)] <-(T,R)
u_next_instruction ! unlock in next instruction
```

The current environment in the Main Processor is not necessarily on the top of the local stack. Before creating either a choicepoint or an environment the Main Processor must determine the top of the local stack. It does so using the `arithoff(MAX,T,TE,T,B,T,E)` instruction. The (T,E) register is then the top of the local stack. The current environment is always on top of the local stack on the Slave Processors since they do not create choicepoints. Choicepoints contain the values of the (G,HP) and (G,TR) registers. The Main Processor must wait until all the previous tasks have completed before it can be sure that the (G,HP) and (G,TR) values are correct. The `waitexec` instruction achieves this by stalling until all previous tasks have completed.

5. A FPPM Implementation

This chapter describes an implementation of FPPM in which each processor (i.e the Main Processor as well as the Slave Processors) may be implemented in a CMOS VLSI chip. Figure 5.1 is an overview of a FPPM system consisting of a Main Processor, 1 to 7 Slave Processors and a shared memory system with separate ports for instruction and data. In addition, FPPM has the following interconnections between the processors to implement special instructions in the FPPM architecture. The Main Processor dispatches tasks to Slave Processors over a *task dispatch bus*. Updates to the global registers and the write-once registers are done over a *result distribution bus*. There are also various control signals including (1) a fail signal for each write-once register set to indicate that the fail instruction has been executed for that set by some processor, (2) a busy line for each write-once set to indicate whether or not there are pending instructions for the set, (3) a free line for each Slave Processor that indicates that the Slave Processor is free, (4) request, grant and release lines for locks, (5) request and grant lines for result distribution bus arbitration and (6) lines to reset the valid bits of all the registers in a register set.

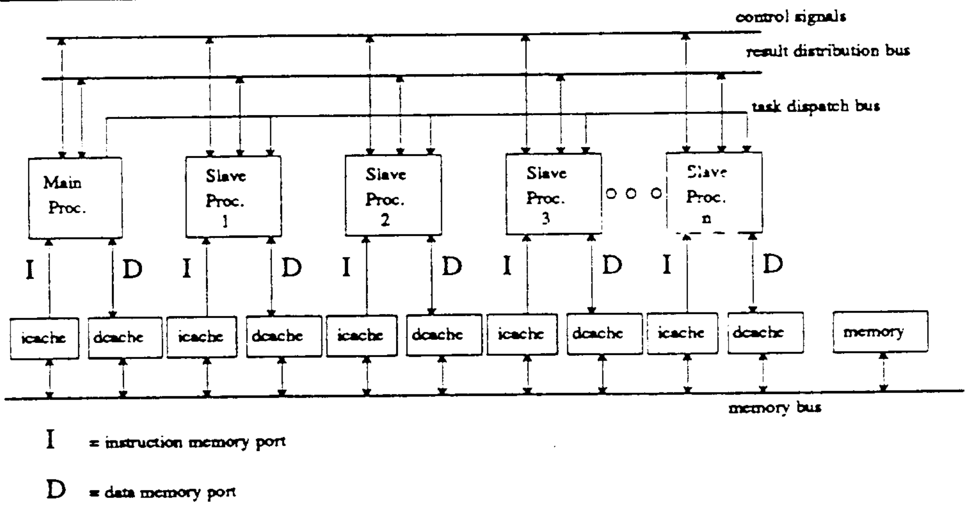


Figure 5.1: Overview of an FPPM implementation. Shared memory is implemented by snooping caches.

The chapter is organized as follows. We describe the clocking scheme is section 5.1. Since the instruction sets of the Main Processor and the Slave Processors are very similar, their data paths are similar as well. We describe both the data paths in section 5.2. We describe the shared memory implementation alternatives in section 5.3. We discuss minimum overhead for parallel execution in section 5.4.

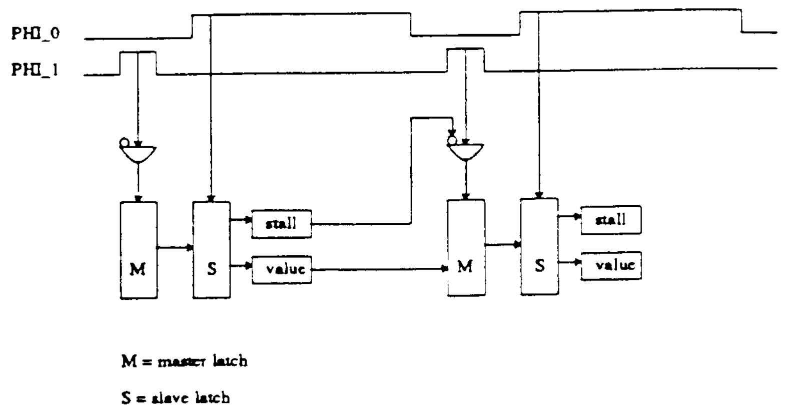


Figure 5.2: Clocking Scheme

5.1. The Clocking Scheme

The implementation of FPPM processors described in this chapter uses a simple 2-phase non-overlapping clock scheme shown in figure 5.2. All changes to the state of the processor are latched during phase 1 (PHI_1) into the master latches. Slave latches are loaded during phase 0 (PHI_0) and all combinational logic is performed between the start of PHI_0 and the beginning of PHI_1. Combinational logic performs two functions in parallel: (1) new values are computed and (2) stall signals are computed. The stall signals are used to inhibit latching of new values into master latches during PHI_1. Stall signals stall pipeline stages in order to handle events that are asynchronous to the Slave Processor such as cache misses, non-availability of result distribution buses and invalid write-once register operands.

5.2. The Processor Data Path

Since the data paths of the the Main Processor and the Slave Processors are very similar, we describe them both in this section. A schematic of data path of a processor is shown in figure 5.3. A more complete description of the processor hardware is the ISP register transfer level description listed in appendix 3.

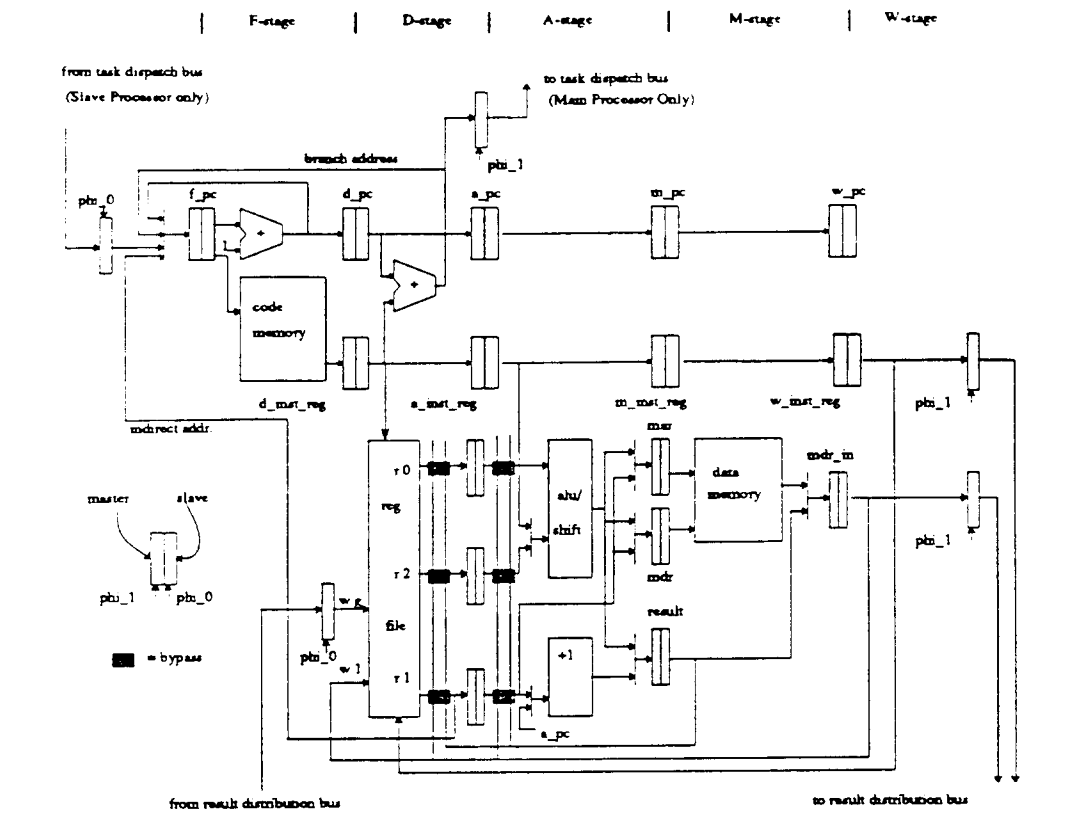


Figure 5.3: The processor data path consists of a program counter pipe, an instruction pipe, and a computation pipe. The register file block in the diagram contains all three types of registers: local, global and write-once. Bypass paths are provided only for local registers. The global and write-once registers are written from the result distribution bus through the write port labeled "wg". the local registers are written through the write port labeled "wl".

The processor pipeline has 5 stages: the fetch stage (f-stage) the decode stage (d-stage), the ALU stage (a-stage), the memory stage (m-stage) and the write back stage (w-stage). Instructions are fetched from the code memory (on-chip instruction cache) in the f-stage. The d-stage decodes the instructions, reads register source operands, computes the branch address, executes branches and arbitrates for the lock. The a-stage has an alu/shifter as well as an incrementer (for *push* and *bral* instructions). The m-stage performs data memory accesses. The w-stage writes the destination local register, arbitrates for the result distribution bus, writes data to the result distribution bus and release the lock. We describe the data path in three sections: the program counter pipe, the instruction pipe and the computation pipe.

5.2.1. The Program Counter Pipe

Each stage has its own program counter (pc). The f-pc is used to fetch the next instruction from the code memory port. The instruction is loaded into the d-stage instruction register, d-inst-reg, and the d-pc is loaded with the instruction's address + 1. The pc then gets transferred along with the instruction down the pipeline. In each stage (other than the f-stage) the pc is 1 greater than the instruction in that stage. The d-pc is used to compute the branch address. In the case of the Main Processor, the branch address is also the address of the task created by the *exec* instruction. In the case of the Slave Processor, the f-pc is loaded from the task dispatch bus when the Slave Processor is free. In the Slave Processor each pc has a valid bit indicating that the instruction in that stage is valid. The valid bits of all the pc's are initially cleared. The valid bits of any stage will be cleared if the fail line for the set number of the instruction in the stage is high. The *done* instruction clears the valid bit of the f-pc. The free line for the Slave Processor is pulled high when the f-pc is invalid.

5.2.2. The Instruction Pipe

The instruction pipe consists of instruction registers for the d-stage, a-stage, m-stage and w-stage. The instruction pipe also contains a register for the set number of the instruction of each stage (these set number are not shown in figure 5.3 for lack of space) and the set number also travels down the pipe along with the instruction. In the case of the Slave Processor the f-stage set number register is loaded from the task dispatch bus along with the address of a new task. In the case of the Main Processor, the set number of the f-stage can be incremented or decremented by the *new* and *old* instructions respectively. If there is a valid instruction in a pipeline stage, then the busy line for the write-once register set is pulled high. Since these busy lines are wired-or, the line for a set will be high if there are any outstanding instructions for the set in any processor.

5.2.3. The Computation Pipe

The d-stage of the computation pipe contains the register file (including local, global and write-once registers) and a register bypass for local registers. The bypass path is required for branch instructions that have local register source operands. The bypasses work as follows. The register file is read during PHI_0. The bypass logic active during PHI_0 compares the source register addresses with destination register addresses of instructions in the a-stage and m-stage. If the a-stage address matches and the register is a local register then the register value is taken from the result register slave latch, else if the m-stage address matches and the register is a local register then the register value is taken from the mdr-in slave latch, else no bypass is performed. Thus, each bypass point shown in figure 5.3 is a 3-input multiplexer and two register address comparators. The a-stage contains a second bypass path for local registers, an ALU/shifter and an incrementer. This second bypass path is required for a-stage local register source operands. The m-stage contains a data memory (discussed in greater detail later). The w-stage arbitrates for the result distribution bus if it has to write a global or write-once register.

The register file contains all three types of registers: local, global and write-once (see figure 5.4). It has two write ports and three read ports†. The implementation has multiple write-once register sets (4 and 8 sets are simulated), but any given instruction can access only two consecutive sets: the input set is addressed by the set number of the instruction and the output set is the next set in circular order. One of the write ports is exclusively for the local registers while the other write port is for the updates to the global and write-once registers over the result distribution bus. The write-once registers have a valid bit each. The valid bits of an entire set are cleared by when the clear signals are set by the Main Processor or when the fail line of the previous set are high. The valid bit for a write-once register is set when the register is written.

5.3. Processor Control

5.3.1. Decoding and Sequencing

FPPM processor instructions are decoded in the d-stage. The control signals are listed in

All three read ports are never used at the same time. In fact, it is possible to change the instruction format so that the two register source operands are always in the same position in the instruction (this is required if register reads are to be done in the decode stage). However, some fields of the instruction will then not be contiguous. Since non-contiguous fields are a hindrance for writing and debugging the simulator, we use three register read ports in the simulator and in figure 5.3. If we were actually building the processor we would use only two read ports

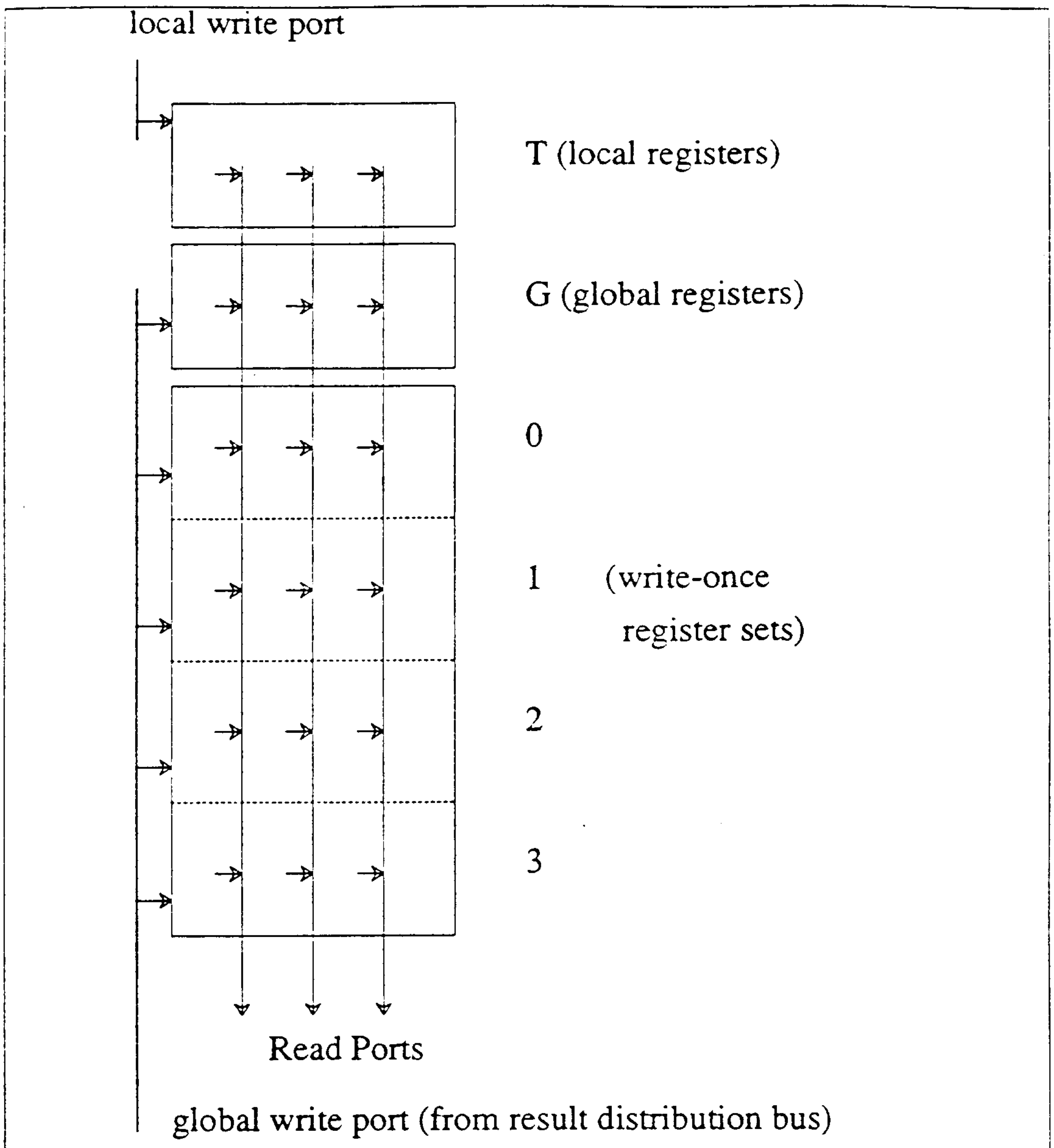

local write port
T (local registers)
G (global registers)
0
1 (write-once register sets)
2
3
Read Ports
global write port (from result distribution bus)

Figure 5.4: The register file contains all three types of registers: local (T), global (G) and write-once (sets 0 through 3 are shown). The read ports are common to all the the register types, but there are two write ports, one for the local registers and one for the global and write-once register sets.

appendix 3. In this section, we only discuss those control signals that could be part of a critical path. Since the d-stage executes the branches (both conditional and unconditional), the control signals for the branches must be made available quickly. These signals control selection of the branch destination and conditions for branches. However, the computation of possible branch destinations as well as all the conditions can proceed *before* the control signals are available: the control signal selects one condition from among the various conditions and one destination from among the various destinations. The possible conditions for branches in the FPPM are such that they can be computed quickly, but they require the source registers to be read first. Although register reads can be performed as soon as the instruction is loaded into the d-inst-reg, the register file access time, the register bypass, the condition computation and the selection of branch destination could be the critical path in the chip. If this is the case, then the following technique may be used to reduce the length of this critical path by reducing the access time for the data required to compute the condition. Most of the conditions use the tag field of the registers. Only two conditions use the value field: GEZ and EQZ. Therefore the critical path can be reduced if additional bits for the GEZ and EQZ conditions and the tags for the register are stored in a separate register file that is faster than the larger register file for the value fields. The BAM processor [40] uses this approach for the tags. The GEZ and EQZ conditions can be computed and stored along with the tags every time a register is written into the register file.

5.3.2. Arbitration, Locks and Stalls

Processor in FPPM have to arbitrate for locks (in the d-stage) and for the result distribution bus (in the w-stage). In each case the request is sent out at the beginning of PHI_0. The grant signals are expected in sufficient time to stall the pipeline stage if the bus or lock is not granted. The pipeline stalls are different for the Main Processor and for the Slave Processors. In the case of the Main Processor, a stall in any pipeline stage stalls the entire pipeline. In the case of the Slave Processor the execution of different tasks are decoupled from each other. A stall in a pipeline stage of an Slave Processor stalls all the previous pipeline stages, but only those of the following stages that are executing instructions from the same task. This is necessary to avoid deadlocks caused by data dependencies between two tasks that execute on the same Slave Processor. The deadlock may occur if the destination of an instruction in the earlier task is a a write-once register that is also the source of an instruction in the later task. The later instruction could stall in the d-stage before the earlier instruction (which is the only one that can release the stall) reaches the w-stage. If the two tasks were not decoupled, then the later instruction would also stall the earlier instruction, creating a deadlock. The compiler could avoid such deadlocks by ensuring that writes to write-once registers are done sufficiently before the end of a task (or that reads of write-once registers are done sufficiently after the start of the task). However, this reduces the utilization of Slave Processors by requiring several *nop* instructions at the end (or the beginning) of tasks.

5.4. The Memory System

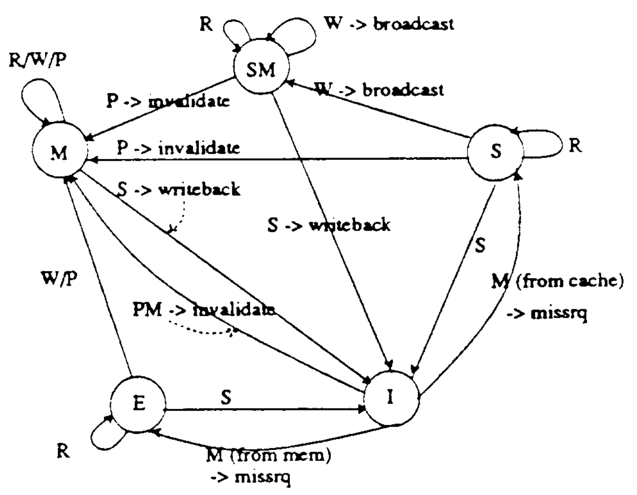
In this section, we discuss practical methods of implementing a shared memory system for FPPM. FPPM processors have separate instruction and data memory ports. Compared to the shared data memory, instruction (code) memory is easy to implement because there are usually no writes to the instruction stream. In this section, we only study the implementation of a shared data memory system.

The most common method method of implementing a high performance shared memory system is using snooping caches [35]. In a snooping cache system, a single shared bus is used for communication. The single shared bus cannot support a large number of caches because of limited bus bandwidth. Cache coherence is maintained in snooping cache systems by means of a cache coherence protocol. There are a large number of cache coherence protocols. A detailed analysis of the tradeoffs involved in each of these coherence schemes is beyond the scope of this dissertation. We only consider a few promising alternatives.

Two of the most important characteristics of the cache protocol are (1) the way in which they maintain consistency between caches and memory (write through versus write back), (2) the way in which they maintain consistency between caches (update versus invalidate). With a write back protocol, modified data is written to memory only when the block is replaced, whereas with a write through protocol, every write is broadcast on the bus. Write through usually results in high bus traffic and, unless some form of write buffer is used, in longer latencies for write operations that occur in quick succession. Therefore we only consider write back protocols. In an update protocol, when a processor writes to a block that is also present in another cache, the write is broadcast over the bus so that subsequent reads of the data in the other cache do not require the bus. In an invalidate protocol, if a processor writes to a block that is also present in another cache, the cache executes a bus transaction to invalidate the block in the other cache so that subsequent writes to the block do not require the bus. Generally, update protocols work better than invalidate protocols when shared data is written only a few times in a cache, but read very often by other caches. Invalidate protocols work better than update protocols when shared data is written often in the same cache, but read only a few times in other caches. In FPPM's flow parallelism, shared data are shared variables. They are written only twice (once to create an unbound variable and once to bind the variable), but are read very often (while the consumer process is waiting for it to be bound). Therefore we expect that for FPPM an update protocol is preferable to an invalidate protocol. Simulation results in chapter 6 show that update protocols do perform better than invalidate protocols for FPPM as expected.

We simulate two cache coherence protocols: an update protocol and an invalidate protocol. Both protocols are simulated for a write back, direct-mapped cache. The state diagram for the update protocol is shown in figure 5.5 and the state diagram for the invalidate protocol is shown in figure 5.6. In each case, the diagram consist of two parts: one for the actions initiated by the processor and the other for the actions on the bus. The processor transaction codes and the cache block states are also listed in figures 5.5 and 5.6. Apart from read and write operations, the cache supports push operations. A push operation is a write to the top of a stack. If there is a cache miss on a write, then the cache first fetches the entire block over the bus before proceeding with the write . However, in the case of a push, the cache does not need to fetch the data for the block provided that the word being written is the first word in the block (since it is on the top of the stack). Since a large fraction of Prolog writes are to the top of a stack, it is useful to provide this special operation. The FPPM processor generates a push operation for *push* or *pushtag* instructions provided that the word is at the beginning of the cache block. This optimization was suggested by Morioka *et al* [47]. In FPPM, heap space is first allocated in chunks and then data is often written to the chunks using *push* of *pushtag* instructions. However, another processor could be writing to data in another chunk above. Thus, it is no longer true that the push occurs only for words on the top of the stack. If the cache did not fetch the block on a miss then there could be an inconsistency. This problem is solved by ensuring that chunks are always allocated in sizes that are multiples of the cache block size and

Processor Side State Diagram



Processor Actions

R = Read

W = Write

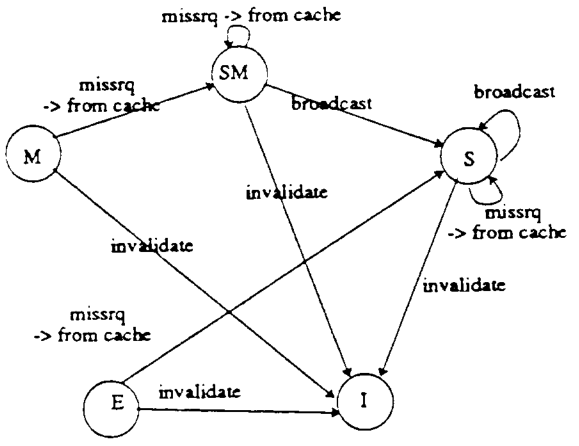
P = Push

M = Miss (read or write)

PM = Miss (push)

S = Swapout

Bus Side State Diagram



Cache States

I = Invalid

E = Exclusive Unmodified

M = Exclusive Modified

SM = Shared Modified

S = Shared Unmodifed

Figure 5.5: State diagram for FPPM's cache coherence protocol (update)

that initially the heap pointer is aligned with the cache block.

The cache block states are Invalid (I), Exclusive Unmodified (E), Exclusive Modified (M), Shared Modified (SM) and Shared Unmodifed (S). The exclusive states (E,M) indicate that the block is not present in any other cache. Writes to an exclusive block do not require the bus. The shared states (S,SM) indicate that the block may also be in another cache. In the case of the update protocol, a write to a shared block must be broadcast over the bus. In the case of the invalidate protocol, a write to a shared block must first invalidate the copies of the block in the other caches. The modified states (M,SM) require that the block be written back to memory when it is swapped out.

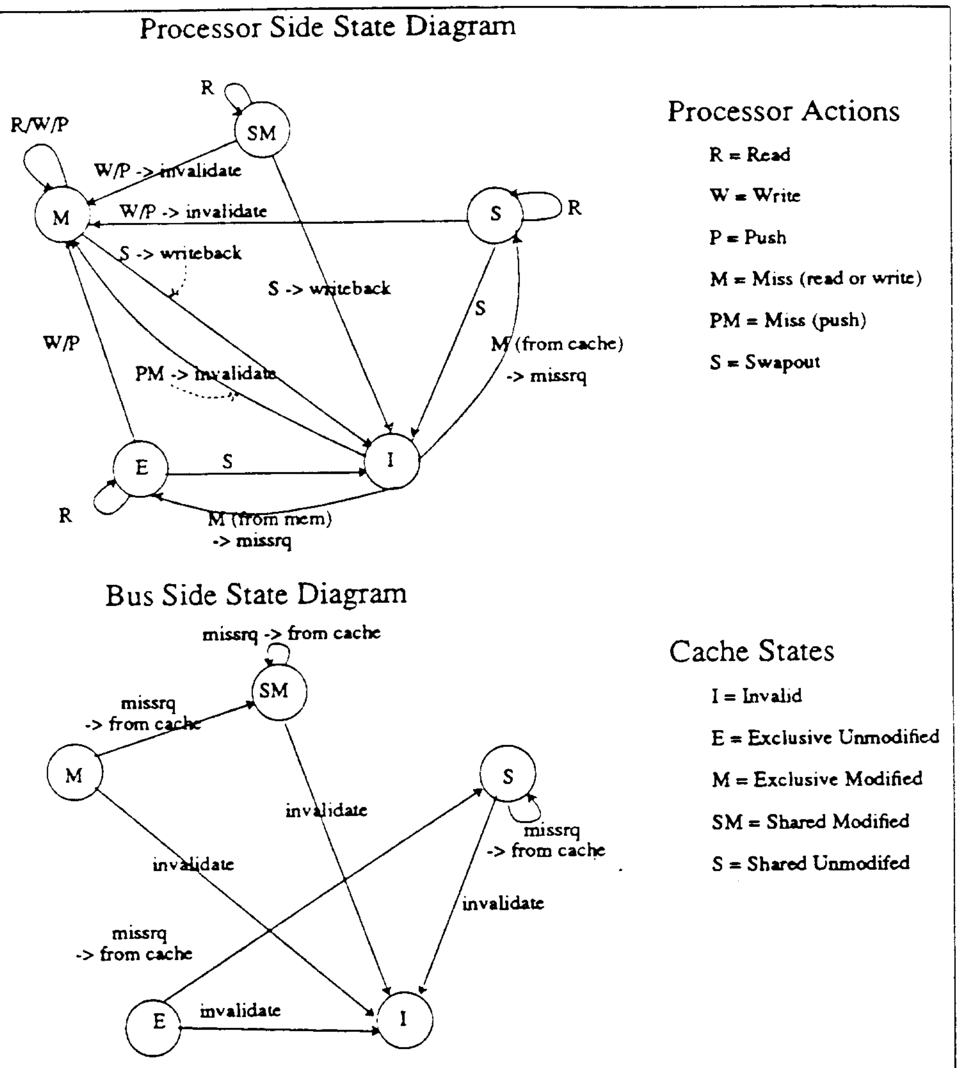


Figure 5.6: State diagram for FPPM's cache coherence protocol (invalidate)

The SM state indicates the the block is both shared and modified. A block can only be in the SM state in one cache; this cache is the one that wrote to the block most recently, and that block has the responsibility of writing it back to memory if it needs to be replaced.

The cache implementation has two shadow copies of the tag and state stores so that the bus and processor transactions can execute simultaneously provided that only one of the transactions needs to write to the tags or state. If both transactions need to write to the state or tag stores then the processor transaction is stalled.

Operation	Latency
hit, no bus transaction required	1
hit, invalidation or broadcast required	3 + s
push miss, no writeback required	3 + s
push miss, writeback required	5 + w + s
read/write miss, no writeback, data from cache	3 + w + s
read/write miss, no writeback, data from memory	4 + w + s
read/write miss, writeback, data from cache	6 + 2w + s
read/write miss, writeback, data from mem	7 + 2w + s

Table 5.1. Latencies for cache operations, where s is the number of unsuccessful bus arbitration cycles and w is the number of bus cycles required to transfer a line after obtaining the bus. In the simulations, we assume that w is equal to the number of words on a cache block.

The latencies of various cache operations is summarized in table 5.1. These latencies assume that there are no bus transactions that write to the state or tag stores and stall the processor transaction.

5.5. Evaluation of Overhead for Parallel Execution

Overhead in FPPM

Parallel task creation in FPPM is very inexpensive. The Main Processor can initiate a parallel task on a Slave processor with a single `exec` instruction. This instruction stalls the Main Processor until a Slave Processor is available. If a Slave Processor is available, the cost of task creation on the Main Processor is a single cycle. The task begins execution on the Slave Processor three cycles later. The arguments of the task are usually in the write-once registers and can be written after the task has been created, either by the main program or by another parallel task. Therefore, the task can begin execution before all its arguments are available.

The heap pointer and trail pointer are stored in global registers in FPPM. Heap or trail space can be allocated in 2 cycles (the pointers remain locked for 6 cycles). If all N arguments required by the task are loaded into write-once registers by the Main Processor and all N arguments are subsequently copied into the local registers by the Slave Processor, the total overhead of task creation is $2N+1$.

Overhead if the Result Distribution Bus or the Task Dispatch Bus are Eliminated

Task creation is more expensive if the special buses (i.e., the result distribution bus and the task dispatch bus) are eliminated. Without these special buses, the Main Processor can communicate with the Slave Processors only through shared memory. The Main Processor creates tasks by appending a task frame (containing the task's arguments and starting address) to a task queue. The Slave Processors remove task frames from the task queue, load the arguments and starting address from the task frame and begin execution at the starting address.

Assuming an ideal multi-port shared memory with single cycle access, the cost of task creation for the Main Processor is $N+4$ cycles, where N is the number of arguments for the task. $N+2$ cycles are for storing the arguments, the size of the task frame and starting address in the task frame, and the additional 2 cycles are for computing and storing the new pointer to the tail of the task queue. Since the Slave Processors share a single task queue, their accesses to the queue must be synchronized. In order to eliminate frequent locking and unlocking of the task queue when it is empty, the Slave Processors poll the status of the queue without locking it while it is empty. The Slave Processors obtain tasks from the queue as follows:

```
get_task:
    repeat
        read queue status
    until (queue not empty)

    lock (queue head)

    read queue status;
    if (queue empty)
    {
        unlock (queue head)
        goto get_task
    }
    else
    {
        remove task from queue
        update queue head
        unlock (queue head)
        execute task
    }
```

The status of the queue is the head and tail of the queue; the queue is empty if the head is equal to the tail. The status of the queue is determined by reading and comparing the head and tail pointers. A task is removed from the queue by reading the size of the task frame and loading the arguments and starting address into registers from the task frame. The queue head is then updated by adding the size of the task frame to the head pointer of the queue and storing the new head pointer in memory. Once a task frame (with N arguments) has been added to the queue by the main processor, a Slave Processor takes a minimum of $N+15$ cycles to remove the task from the queue and begin executing it. This assumes that the memory is an ideal multi-port shared memory, that there are no stalls to obtain the lock, and that the Slave Processor executes the `get_task` code sequence as soon as the Main Processor has added the task to the task queue. Therefore, the minimum total overhead for task creation using a task queue in ideal shared memory is $2N+19$ cycles. If shared memory is implemented by snooping caches, the additional overhead for the shared task frame is at least $N+5$ cycles (3 cycles latency to initiate a bus transfer, and 1 cycle each for transferring the $N+2$ entries in the task frame). Since the head and tail pointers are also shared among the processors, each shared access to the these pointers results in a bus transfer.

Without a Result Distribution Bus, there can be no global registers. Consequently the heap and trail pointers must be stored in memory locations. Assuming an ideal multi-port shared memory, heap and trail space allocation now require 4 cycles (the pointers remain locked for 7 cycles).

Overhead in the Unrestricted Scheme

In the unrestricted scheme, processors queue for tasks (instead of tasks queuing for processors). This scheme requires synchronization for both ends of the queue: at the head when acquiring a processor to execute a parallel task, and at the tail when adding a processor to the queue. However, the advantage of this scheme is that parallel tasks may create other parallel tasks. If a processor is available in the queue, then the task is dispatched to the processor, otherwise the task is done sequentially.

We consider two cases: (1) with global registers and a result distribution bus and (2) with no shared registers or result distribution bus. In both cases a parallel task's arguments and starting address are stored in shared memory. However, in (1) the shared pointers (heap, trail, processor queue head head and processor queue tail) are stored in global registers, whereas in (2) the shared pointers are stored in memory. The overhead for these cases is discussed below.

(1) The overhead of checking whether a processor is available to execute a parallel task is 2 cycles. This overhead is incurred even if no processor is available and no parallel task is created. The minimum additional overhead of removing a processor from the queue is 3 cycles (the head of the queue remains locked for 7 cycles). Assuming an ideal multi-port shared memory, the cost of dispatching the task to the processor (i.e., loading the arguments and starting address of the task into a frame in shared memory) is $N+1$ cycles. The processor incurs a minimum additional overhead of $N+7$ cycles before it can begin executing the task: 6 cycles to determine that its frame in memory has been loaded with a task, and $N+1$ to load the task's arguments and starting address into its registers. Having completed the task, the processor incurs an minimum overhead of 4 cycles (the tail of the queue remains locked for 6 cycles) to place itself in the queue. Therefore, the minimum total overhead for task creation using a processor queue in ideal shared memory is $2N+17$ cycles.

(2) Additional overhead over Case (1) is incurred if the shared pointers are stored in shared memory instead of shared registers. The minimum overhead of checking whether a processor is available is 6 cycles. The minimum additional overhead of removing a processor from the queue is 6 cycles (the head of the queue remains locked for 10 cycles). The overhead for the processor to load the arguments and starting address are the same ($N+7$ cycles). Having completed the tasks, the processor incurs a minimum additional overhead of 6 cycles (the tail of the queue remains locked for 7 cycles) to place itself on the processor queue. Therefore, the minimum total overhead for task creation using a processor queue in ideal shared memory is $2N+25$ cycles.

6. Evaluation of Performance and Tradeoffs

In this chapter, we evaluate the design choices and overall performance of FPPM using measurements obtained from a register-transfer level simulator. The FPPM simulator is written using Zycad Corporation's ISP hardware description language and simulation tools [74]. The simulator models the architecture at a register transfer level with a Main Processor and up to 7 Slave Processors. The simulator also models several shared memory systems including (1) an ideal multiport memory whose access time for each port can be preset, (2) a snooping cache memory system using an update protocol and (3) a snooping cache system using an invalidate protocol.

Benchmark	Description
Benchmarks that do not create choicepoints	
fib7	Computes the 7th number in the Fibonacci series
hanoi8	Towers of Hanoi problem for 8 disks
nrev30	Naive reverse of a list of 30 elements
qsort50	Quicksort of a list of 50 integers
qsort50r	Same as above with first 2 integers in list interchanged
tak963	Takeuchi function tak(9,6,3,X)
vadd*n*	Add a vector of size *n* to each row of an *nxn* matrix
gennrev*n*	Creates a list of *n* elements and reverses it
Benchmark that creates choicepoints frequently	
queens4	Computes first solution to the 4 queens problem

Table 6.1: The benchmarks programs

The benchmark programs are listed in table 6.1. Since FPPM only exploits parallelism between choicepoints, we consider mainly programs that do not create choicepoints. The performance of programs that have only a small amount of computation between choicepoints, such as the `queens4` benchmark, cannot be improved significantly by FPPM. The benchmarks that do not create choicepoints were chosen as representatives of the various types of inner loops that one might find between choicepoints in a program. The `nrev30`, `vadd`*n* and `gennrev`*n* benchmarks are examples of nested linearly recursive loops. The `nrev30` benchmark was chosen because it is a well-known Prolog benchmark whose performance has been measured or simulated on most Prolog systems. The `gennrev`*n* benchmark is similar to `nrev30`, but it allows us to measure the performance as a function of the size of the list. There are close knit dependencies between the inner loops of the `gennrevn` and `nrev30` benchmarks. The `vaddn` benchmark has no data dependencies between the inner loops. Therefore the `vaddn` benchmark can be expected to perform very well. The remaining benchmarks contain non-linearly recursive predicates. Non-linearly recursive predicates are commonly used for the divide-and-conquer type of algorithms. The `tak963` benchmark has a clause with 4 recursive goals (i.e., its degree of recursion is 4), whereas all the others have clauses with 2 recursive calls (i.e. their degree of recursion is 2). In the `hanoi8` and `fib7` benchmarks the various recursive goals have few data dependencies between them, whereas the recursive goals of `qsort50` and `qsort50r` benchmarks have close knit dependencies between them.

Throughout this chapter we list both arithmetic means and geometric means of performance ratios, but we only use the geometric means for performance comparisons. This chapter summarizes the results of our measurements. The detailed measurements and graphs are given in appendix 1.

6.1. FPPM's Sequential Performance

When evaluating FPPM's speedup due to parallel execution we compare the parallel execution time with sequential execution time on a single FPPM processor (the Main Processor). Therefore, it is important to show that FPPM's sequential execution time is competitive with other high performance sequential Prolog processors. In this section, we compare execution time of a single FPPM processor with the Berkeley VLSI-PLM [60] (a high performance WAM-based processor) and the VLSI-BAM [40] which is the highest performance sequential Prolog processor that we are aware of.

Table 6.2. lists the number of execution cycles for the VLSI-PLM, VLSI-BAM and a single FPPM processor. The cycle time of the VLSI-PLM is 100 nsec. The cycle time of the VSLI-BAM is expected to be 33 nsec. We expect that FPPM can also be implemented with a cycle time of about 33 nsec. The last two columns are ratios of cycles. The VLSI-BAM executes benchmarks in approximately 4 times fewer cycles than the VSLI-PLM. Sequential FPPM performance is roughly equivalent to that of the VLSI-BAM. The differences in performance in table 6.2 are due to the

Benchmark	VLSI-PLM cycles	VLSI-BAM cycles	FPPM cycles	VLSI-PLM/ VLSI-BAM	VLSI-BAM/ FPPM
fib7	6647	1447	754	4.59	1.92
hanoi8	72784	17974	8179	4.05	2.20
nrev30	21122	4218	4801	5.01	0.88
qsort50	43121	6175	5901	6.98	1.05
queens4	2819	1355	1380	2.08	0.98
tak963	43275	5351	3374	8.09	1.59
arith. mean	-	-	-	5.13	1.43
geom. mean	-	-	-	4.71	1.35

Table 6.2: Performance Evaluation of Sequential Code on FPPM. The VLSI-PLM and the VLSI-BAM figures are for compiled programs whereas the sequential FPPM programs are written in assembly.

following two reasons:

(1) FPPM code includes additional optimizations performed manually. These optimizations can reasonably be expected of a compiler, but have not yet been implemented by the VLSI-BAM compiler. As more optimizations are implemented by the compiler, we expect the VLSI-BAM performance to be closer to the sequential FPPM performance.

(2) The FPPM and BAM instruction sets are similar, but not identical. One of the more important differences is in the memory access instructions. BAM has instructions to load and store two words at a time over a 64-bit data bus to the cache whereas FPPM only does single word memory operations. However, the VLSI-BAM takes 2 cycles for memory writes whereas the FPPM assumes that writes can occur in one cycle.

Table 6.2 shows that FPPM's sequential execution time is competitive with the most efficient sequential implementations. Consequently, the speedups over the sequential execution time that we measure are true measures of the benefits of parallel execution in FPPM.

6.2. Speedup Due to Parallel Execution in FPPM

We have just shown that using a single FPPM processor one can achieve performance competitive with other fast sequential Prolog processors. In this section, we estimate the speedup that can be obtained by exploiting fine grain parallelism in FPPM. For the measurements in this section, we initially assume an ideal shared memory (i.e. a multiport memory with single cycle reads and writes at each port). Clearly, this assumption is unrealistic, but we wanted to get an idea of the performance that we could achieve if we had such an ideal memory. Measurements of more realistic shared memory systems will be presented later in this chapter.

Bench-mark	Sequent. (cycles)	Number of Slave Processors						
		1	2	3	4	5	6	7
fib7	754	0.82	1.41	1.70	1.70	1.70	1.70	1.70
hanoi8	8179	0.93	1.82	1.83	1.83	1.83	1.83	1.83
nrev30	4801	0.70	1.22	1.61	1.90	2.11	2.28	2.38
qsort50	5901	0.84	1.18	1.18	1.18	1.18	1.18	1.18
qsort50r	5901	0.84	1.38	1.63	1.81	1.81	1.81	1.81
tak963	3374	1.00	1.83	1.83	1.83	1.83	1.83	1.83
vadd16	7585	0.89	1.72	2.56	3.27	3.94	4.70	5.18
arith. mean	-	0.86	1.51	1.76	1.93	2.06	2.19	2.27
geom. mean	-	0.86	1.49	1.72	1.85	1.93	2.01	2.05

Table 6.3: Speedup relative to sequential execution due to fine grain parallelism in deterministic programs. Ideal multi-port shared memory with 1 cycle access and FPPM implementation with 8 write-once register sets are assumed.

We saw, in chapter 3 that FPPM's execution is serialized by choicepoint creation or side-effects and that parallelism can be exploited only between choicepoints or side-effects. Consequently, we estimate the overall speedup of FPPM by analyzing FPPM execution in two parts in the sub sections below. We first estimate the speedup of programs that do not create choicepoints or side-effects. Next, we show that choicepoint creation and backtracking with parallel execution do not contribute large overhead by examing FPPM's execution of a program that has relatively little computation between choicepoints.

6.2.1. Speedup for Predicates that Do Not Create Choicepoints or Cause Side-Effects.

Since execution is serialized by choicepoints, we first measure the speedup obtained by predicates that do not create choicepoints. The speedup of these predicates indicates the speedups that can be obtained between choicepoint creations. Table 6.3. lists the sequential execution time (in cycles) and speedup relative to the sequential execution time for a number of benchmarks that do not create choicepoints. Speedups are listed for FPPM with 1 through 7 Slave Processors and 8 write-once register sets.

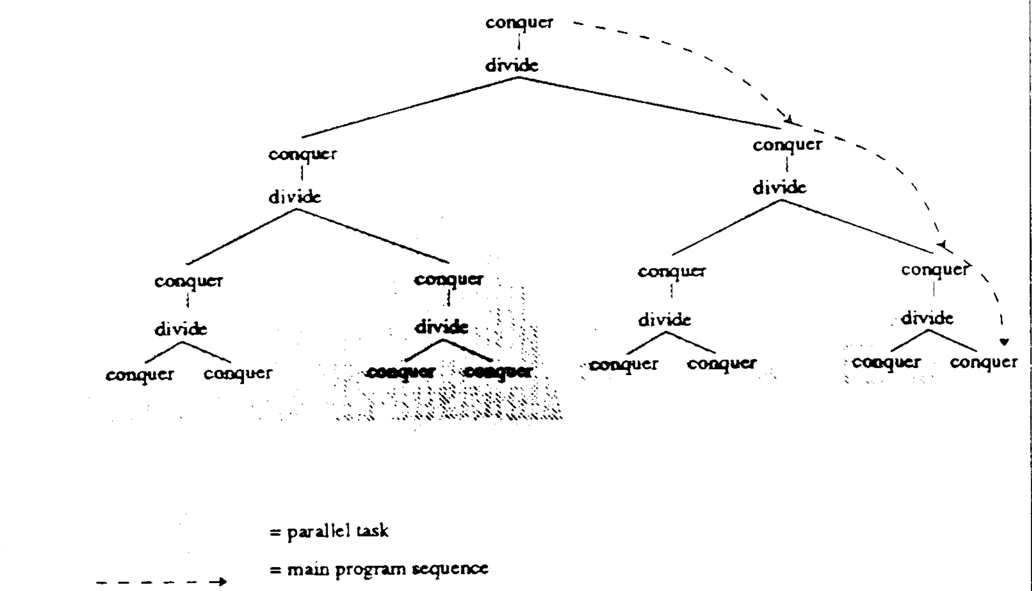


Figure 6.1: Partitioning of divide-and-conquer problem in FPPM

The largest speedup (5.18) is obtained for the `vadd16` benchmark, in which there are few data dependencies and the size of each parallel task is the same. A good speedup is also obtained for the `nrev30` benchmark. The `vadd16` and `nrev30` benchmarks are linearly recursive whereas the others are non-linearly recursive. Non-linearly recursive benchmarks often experience less speedup than linearly recursive predicates because these benchmarks use divide-and-conquer algorithms and FPPM creates a parallel task for an entire partition as shown in figure 6.1. If the the problem is divided into two equal parts in the first divide stage, then the parallel task that is created for one partition does almost half the work of the entire program. The performance depends on the size of the partitions in each case. The `qsort40` benchmark's performance is poor because the first parallel task created corresponds to a large partition. The `qsort40r` benchmark's performance is better because the first parallel task created corresponds to a smaller partition.

The speedups in table 6.3 are also limited by the fact the at the beginning and end of the execution of each program most of the Slave Processors are idle. The effect of this is illustrated by the `nrevnrev` benchmark in which a list is reversed and then reversed again using the `nrev/2`

Bench-mark	Sequent. (cycles)	Number of Slave Processors						
		1	2	3	4	5	6	7
nrevnrev	9545	0.69	1.24	1.64	1.96	2.19	2.37	2.48

Table 6.4: Speedup relative to sequential execution due to fine grain parallelism in the `nrevnrev` program, which executes the `nrev/2` predicate of the `nrev30` benchmark on a list and then applies the same predicate to the resulting list. Ideal multi-port shared memory with 1 cycle access and FPPM implementation with 8 write-once register sets is assumed.

predicate defined in the `nrev30` benchmark. Table 6.4 lists the speedups of the `nrevnrev` benchmark. As expected, the speedup of `nrevnrev` is greater than that of `nrev30`.

6.2.2. Performance Impact of Frequent Choicepoint Creation

Choicepoint creation under parallel execution results in some extra overhead compared to sequential execution. In this section, we measure the effect of frequent choicepoint creation with very little parallel execution between choicepoints. The `queens4` benchmark has these characteristics. Table 6.5 lists the speedup of the benchmark as a function of the number of Slave Processors. The performance of FPPM drops by very little for programs that have frequent choicepoint creation and very little parallel execution between choicepoints.

Bench-mark	Sequent. (cycles)	Number of Slave Processors						
		1	2	3	4	5	6	7
queens4	1380	0.83	0.98	0.98	0.98	0.98	0.98	0.98

Table 6.5: Speedup relative to sequential execution for a program with frequent choicepoint creation and very little parallel execution between choicepoints. Ideal multi-port shared memory with 1 cycle access and FPPM implementation with 8 write-once register sets is assumed.

6.3. Effect of Memory Access Time on Performance

The previous section measured speedup due to parallel execution in FPPM assuming an ideal multi-port memory with 1 cycle access. Although it is possible to build multi-port memories, they are slow and expensive. In this section, we measure the performance degradation of FPPM for multi-port memories with longer access times. We only consider access times for the data memory ports. Multiport memories for instructions are easier to construct than data memory ports because

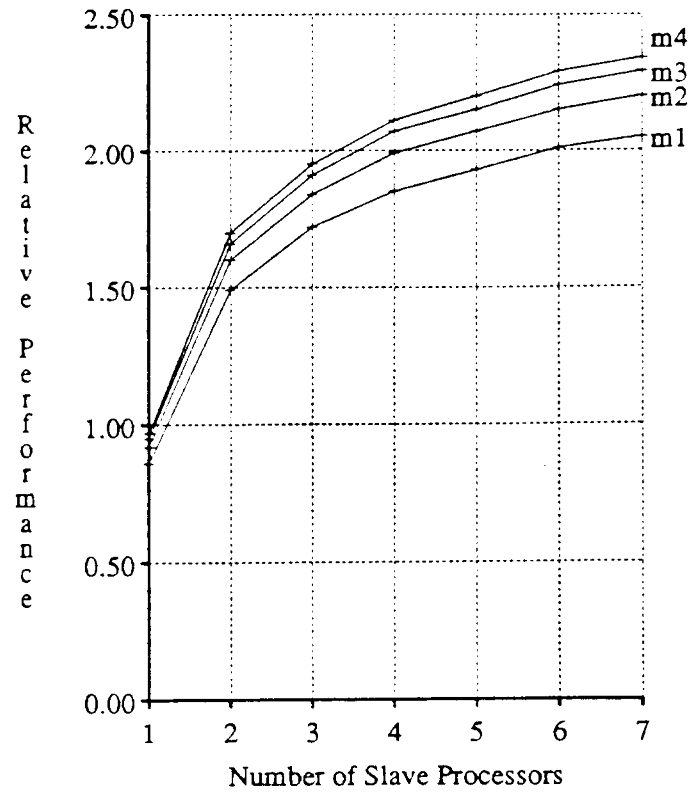


Figure 6.2: Effect of memory access time on the geometric mean performance of FPPM relative to sequential execution with the same memory access time. A FPPM implementation with 8 register sets is assumed.

we assume that there are no writes to the instruction stream. Figure 6.2 plots the geometric mean performance of FPPM (relative to sequential execution with the same memory access time) for memory access times ranging from 1 cycle to 4 cycles. Notice that the speedup due to parallelism increases as memory access time increases. This shows that the performance of parallel FPPM execution is affected less than performance of sequential FPPM execution by increasing memory access time. In figure 6.3 we plot the the geometric mean performance of FPPM relative to sequential execution with 1 cycle memory access as a function of the number of Slave Processors. In figure 6.3 sequential program execution (using only the Main Processor) is represented by data points for 0 Slave Processors. Figure 6.3 shows that performance with parallel execution with 4 or more Slave Processors and 3 cycle memory access is better than sequential execution with 1 cycle memory access.

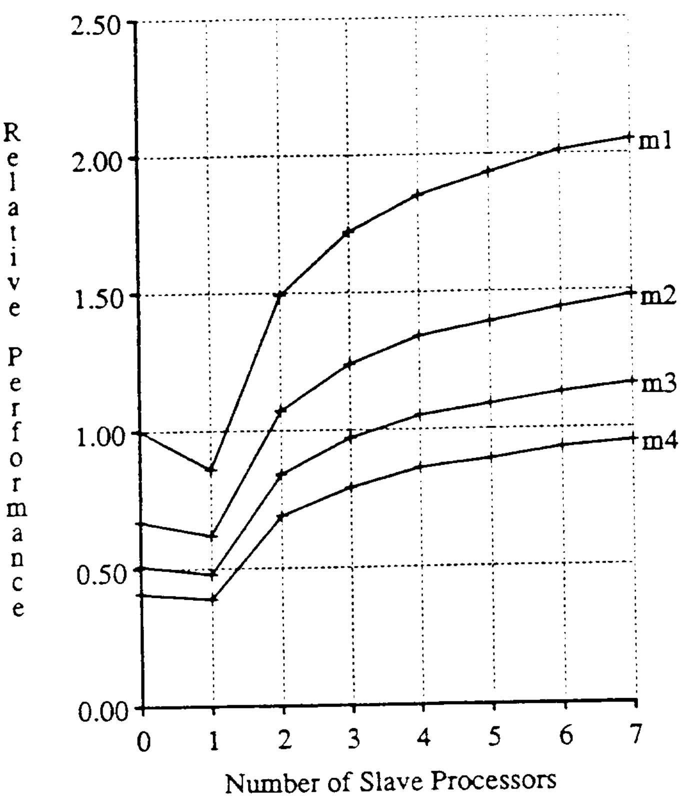


Figure 6.3: Effect of memory access time on the geometric mean performance of FPPM relative to sequential execution with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. A FPPM implementation with 8 register sets is assumed.

The performance graphs for individual benchmark programs are shown in figure A1.1 in appendix 1. The graphs in figures 6.2 and 6.3 were computed using data listed in table A1.1 in appendix 1.

6.4. FPPM's Performance with Snooping Caches

A common method of implementing shared memory systems (for small numbers of processors) is using snooping caches. In this section, we estimate the performance of FPPM with various snooping cache implementations for the data memory ports. As in the previous section, we assume that instruction memory accesses are done in one cycle. We investigate FPPM's performance for direct-mapped caches with the the two write-back cache coherence protocols described in chapter 5: the update protocol and the invalidate protocol. Table 6.6 lists the geometric mean performance of FPPM relative to sequential execution using a uniprocessor cache of the same size and organization,

Protocol	Line Size	Number of Slave Processors						
		1	2	3	4	5	6	7
update	2	0.84	1.34	1.53	1.59	1.70	1.73	1.73
update	4	0.86	1.40	1.59	1.72	1.76	1.81	1.80
update	8	0.89	1.42	1.62	1.74	1.77	1.81	1.80
invalid.	2	0.84	1.33	1.51	1.63	1.64	1.67	1.67
invalid.	4	0.86	1.37	1.55	1.68	1.69	1.72	1.71
invalid.	8	0.89	1.39	1.53	1.64	1.62	1.64	1.65

Table 6.6: Geometric mean performance of FPPM for various cache organizations and coherence protocols relative to sequential execution with the same cache organization. The size of each cache is 8K data words. For sequential execution we simulate a cache of the same size and organization, but without the overhead of a coherence protocol. A FPPM implementation with 8 register sets is assumed.

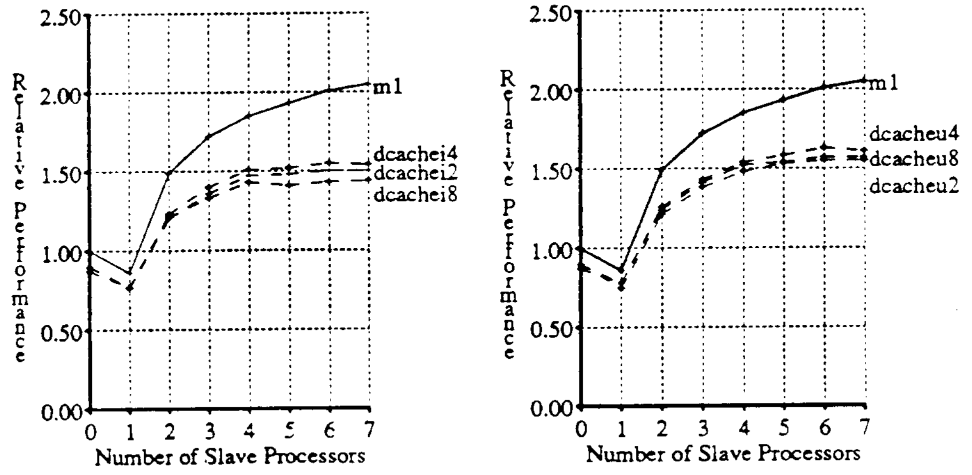


Figure 6.4: Geometric mean performance of FPPM with data caches relative to ideal sequential performance with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. No coherence protocol is used for the sequential case. A FPPM implementation with 8 register sets is assumed. The lines titled "dcache*ps*" are for data caches where *p* is the protocol (i is invalidate, u is update) and *s* is the line size. The m1 line, representing performance with ideal 1-cycle multi-port memory, is drawn for reference.

but without the overhead of a cache coherence protocol. In each case the total size of the cache is fixed at 8K data words (i.e. line size x number of lines = 8096). Performance measures for each individual benchmark are listed in table A1.2 and table A1.3 in appendix 1. We see from table 6.6 that the update protocol performs better than the invalidate protocol.

In figure 6.4 we plot the geometric mean performance of FPPM relative to sequential execution with 1 cycle memory access as a function of the number of Slave Processors. As in figure 6.3, sequential execution (using only the Main Processor) is represented by data points for 0 Slave Processors. Similar graphs for each individual benchmark are given in figure A1.2 in appendix 1.

6.5. Evaluation of FPPM's Specialized Resources

FPPM has a number of specialized resources. The most expensive of these are (1) the specialized buses including the task dispatch bus and the result distribution bus, and (2) the multiple write-once register sets. The costs of these resources are summarized in table 6.7 (chip area for the register fileis based on the register file design for the VLSI-BAM for a 1.2 micron CMOS process). Eliminating these specialized resource entirely results in a model of parallel execution in which all communication between processors occurs through shared memory. The overhead of task creation and communication in such a model is high compared to the overhead in FPPM. In this section, we study the impact on performance of reducing or eliminating these resources.

Resource	Cost
Task dispatch bus	36 pins
Result distribution bus	41 pins
Merged task dispatch/result distribution bus	42 pins
Write-once register file	16 sq. mm.

Table 6.7: Cost of FPPM's specialized resources. These costs assume an implementation with 8 register sets and 16 registers/set. Reducing the number of sets or the number of registers/set results in lower chip area as well as fewer pins for the buses.

6.5.1. Merging the Task Dispatch and Result Distribution Buses

The task dispatch bus and the result distribution bus are expensive resources because they use input/output pins on the processor packages. It is possible to reduce the cost of the implementation by using a single bus for both task dispatch and result distribution. In this section we investigate the performance degradation due to using a single bus instead of separate buses.

The utilization of the result distribution and task dispatch buses depends on the relative size of parallel tasks; if the tasks are small the buses will be more heavily used. table 6.8 lists the fraction

Bench-mark	Number of Slave Processors						
	1	2	3	4	5	6	7
fib7	0.94	0.90	0.87	0.87	0.87	0.87	0.87
hanoi8	0.99	0.98	0.98	0.98	0.98	0.98	0.98
nrev30	0.96	0.94	0.92	0.90	0.89	0.88	0.87
qsort50	0.99	0.99	0.99	0.99	0.99	0.99	0.99
qsort50r	0.99	0.99	0.99	0.98	0.98	0.98	0.98
tak963	1.00	1.00	1.00	1.00	1.00	1.00	1.00
vadd16	0.98	0.96	0.93	0.92	0.90	0.88	0.87
queens4	0.86	0.79	0.79	0.79	0.79	0.79	0.79
arith	0.96	0.94	0.93	0.93	0.93	0.92	0.92
geom	0.96	0.94	0.93	0.93	0.92	0.92	0.92

Table 6.8: Fraction of time that the result distribution bus is idle. FPPM with 8 register sets and ideal multi-port memory with 1 cycle access is assumed.

Benchmark	Number of Parallel Tasks
fib7	19
hanoi8	9
nrev30	61
qsort50	10
qsort50r	12
tak963	4
vadd16	34
queens4	24

Table 6.9: Number of parallel tasks created. This is also the number of cycles that the task dispatch bus is busy during the execution of the benchmark.

of time that the result distribution bus is idle. We see that the bus is idle for 92% of the time on an average. The task dispatch bus is idle even more than the result distribution bus since it is only used once for every parallel task created. The number of parallel tasks created for each benchmark is listed in table 6.9. If the result distribution bus is also used for dispatching tasks and priority is given to task dispatch in the arbitration for bus, then the execution time would increase, in the worst case, by the number of tasks created. The increase in execution time is small for all the benchmarks.

Bench-mark	Sequent. (cycles)	Number of Slave Processors						
		1	2	3	4	5	6	7
8 register sets								
fib7	754	0.82	1.41	1.70	1.70	1.70	1.70	1.70
hanoi8	8179	0.93	1.82	1.83	1.83	1.83	1.83	1.83
nrev30	4801	0.70	1.22	1.61	1.90	2.11	2.28	2.38
qsort50	5901	0.84	1.18	1.18	1.18	1.18	1.18	1.18
qsort50r	5901	0.84	1.38	1.63	1.81	1.81	1.81	1.81
tak963	3374	1.00	1.83	1.83	1.83	1.83	1.83	1.83
vadd16	7585	0.89	1.72	2.56	3.27	3.94	4.70	5.18
arith. mean	-	0.86	1.51	1.76	1.93	2.06	2.19	2.27
geom. mean	-	0.86	1.49	1.72	1.85	1.93	2.01	2.05
4 Register Sets								
fib7	755	0.82	1.38	1.38	1.38	1.38	1.38	1.38
hanoi8	8179	0.93	1.62	1.62	1.62	1.62	1.62	1.62
nrev30	4801	0.70	1.22	1.58	1.58	1.58	1.58	1.58
qsort50	5901	0.84	1.17	1.17	1.17	1.17	1.17	1.17
qsort50r	5901	0.84	1.36	1.60	1.77	1.77	1.77	1.77
tak963	3374	1.00	1.83	1.83	1.83	1.83	1.83	1.83
vadd16	7585	0.89	1.72	2.49	2.49	2.49	2.49	2.49
arith. mean	-	0.86	1.47	1.66	1.69	1.69	1.69	1.69
geom. mean	-	0.86	1.45	1.62	1.65	1.65	1.65	1.65

Table 6.10: Speedup relative to sequential execution for FPPM with 8 and 4 register sets. Ideal, multi-port memory with 1 cycle access assumed.

6.5.2. The Number of Write-Once Register Sets

In general, a new write-once register set is allocated for each new procedure in the main program. If the new register set is not yet available then the main program must stall. Therefore, the main program will stall more often if the implementation has fewer register sets. If one new task is issued for each register set, then the number of register sets also determines how many Slave Processors can be used effectively. If multiple tasks are issued for each register set, then more Slave Processors that can be used effectively. In table 6.10 we compare the performance of FPPM with 4 and 8 register sets. With 4 register sets, FPPM achieves very little performance improvement with more than 3 Slave Processors. With 8 register sets, FPPM's performance continues to improve up to 7 Slave Processors. However, even with 8 register sets, some benchmarks do not benefit from more than 3 Slave Processors.

6.5.3. Eliminating the Task Dispatch and Result Distribution Buses

Eliminating the result distribution bus implies that we also eliminate the shared registers (the global registers as well as the write-once registers). In this case FPPM processors can share data only through the shared memory. This approach has additional overhead due to:

(1) The pointers to the top of the shared heap and trail stacks reside in memory and a processor must obtain a lock before allocating stack space.

(2) The Main Processor dispatches tasks to Slave Processors by enqueuing the task's starting address and its arguments in memory. The Slave Processors must obtain a lock before taking a task from the queue. This increases the overhead of creating a parallel task.

Eliminating the write-once registers has one advantage: Execution does not stall when a new register set cannot be allocated since it is still being used.

A consequence of the extra overhead of creating a parallel task without shared registers or a task dispatch bus is that parallel tasks need to be larger for any performance benefit. In order to see how large a task must be in order to be useful, we run two versions of the `gennrev`*n* and the `vadd`*n* benchmarks for *n* values of 1, 2, 4, 8, 16, 32 and 64. The first version uses the task dispatch bus, result distribution bus and shared registers. These resources are eliminated for the second version and all communication between processors is through shared memory. In figure 6.5 we plot the maximum speedup of each version the benchmarks versus the logarithm (base 2) of the value of *n* for ideal shared memory and for shared memory implemented with snooping caches. More detailed performance plots as a function of the number of Slave Processors for each value of *n* are given in figure A1.3 in appendix 1.

Figure 6.5 shows that FPPM with a task dispatch bus and a result distribution bus achieves better performance for small task sizes. The performance difference is even greater if shared memory is implemented using snooping caches†. As the task size is increased, the difference in

† In figure 6.5 the maximum speedup for both benchmarks for small values of *n* is greater with snooping caches than with ideal memory. This is because the initial memory accesses to the heap and local stack collide in the single, direct-mapped cache of the sequential processor whereas they do not collide for the parallel execution because they occur on different processors. Therefore, the sequential performance is poor for small values of *n* and the relative speedup for parallel execution (with or without the task dispatch and result distribution buses) is high.

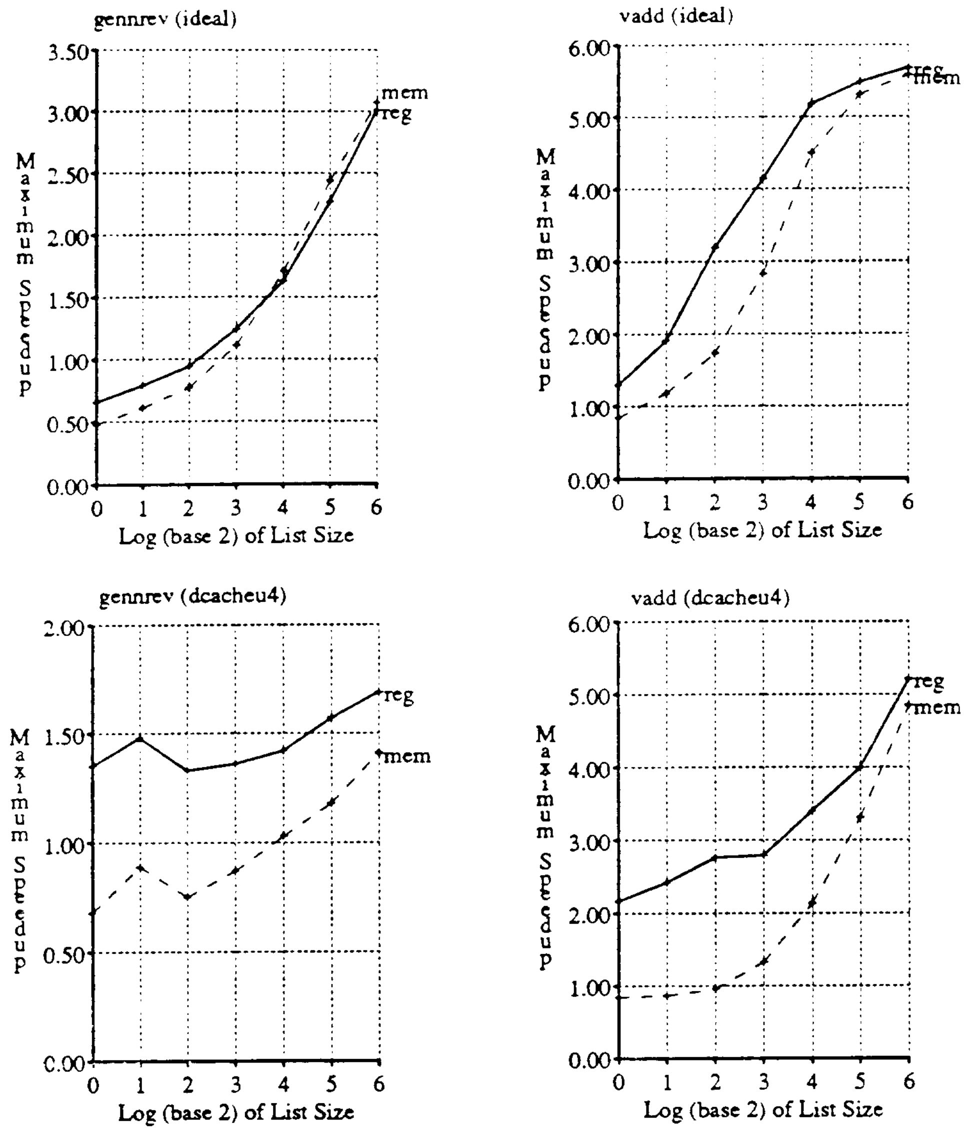


Figure 6.5: Comparison of FPPM's performance for the `gennrev`*n* and the `vadd`*n* benchmarks with and without the special buses (i.e., the task dispatch bus and the result distribution bus). The solid lines labeled *reg* represents the performance with special buses and the dashed lines labeled *mem* represents performance with special buses. The two graphs on top are for ideal, multi-port shared memory with 1 cycle access, whereas the graphs at the bottom are for shared memory implemented using 8K word, direct-mapped, snooping caches with 4 words/line and the update protocol. Performance is plotted relative to sequential execution with a similar memory. The average size of the tasks increase along the x-axis. The graphs illustrate the performance benefits of the special buses for small tasks.

speedup is smaller because the overhead of creating tasks is smaller compared to the total size of the task. For the `gennrev`*n* benchmark the performance is better without the write-once registers for large values of *n* and ideal memory because stalls due to unavailable register sets are eliminated.

6.6. Evaluation of the Alternative (Unrestricted) Model

So far, we have considered the performance of FPPM using a parallel execution model in which only the main program is allowed to create parallel tasks. As explained earlier, such a model limits the amount of parallelism that can be exploited in non-linearly recursive predicates. In chapter 3, we proposed an alternative program partitioning and task scheduling scheme (the Unrestricted Model) that allows parallel tasks to create other parallel tasks. In this section, we compare the performance of the FPPM Model with that of the Unrestricted Model.

	Ideal Memory (1 cycle)			Snooping Cache		
Benchmark	FPPM (cycles)	Unrest. (cycles)	Unrest./ FPPM	FPPM (cycles)	Unrest. (cycles)	Unrest./ FPPM
fib7	443	465	1.05	503	873	1.74
hanoi8	4478	2046	0.46	5301	3699	0.70
nrev30	2018	2512	1.24	3635	4912	1.35
qsort50	4991	2491	0.50	5518	4216	0.76
qsort50r	3263	2646	0.81	3778	4461	1.18
tak963	1845	2845	1.54	1865	4388	2.35
vadd16	1464	2223	1.52	2452	3340	1.36
arith. mean	-	-	1.02	-	-	1.35
geom. mean	-	-	0.92	-	-	1.25

Table 6.11: Performance comparison of the FPPM Model and the Unrestricted Model. All measures are for Configurations with 7 Slave Processors. The snooping cache measurements are for a direct-mapped, 8 Kword cache using the update protocol and 4 words/line. The mean performance of the two models are roughly equivalent for ideal memory. However, if snooping caches are used, the FPPM model performs better by a factor of 1.25. As expected, the FPPM model is better for the benchmarks with linearly recursive predicates (i.e., `nrev30` and `vadd16`). It is interesting to note that the Unrestricted Model does not always perform better then the FPPM Model for non-linearly recursive predicates.

Table 6.11 lists the performance of the FPPM Model and the Unrestricted Model (with 7 Slave Processors for both Models), for two memory systems: ideal multi-port memory and shared data memory implemented by a snooping cache (8 KWords, direct-mapped, update protocol, 4 words/line). The two models have roughly equivalent mean performance if an ideal shared memory is assumed. However, the FPPM Model has better mean performance for shared memory implemented with snooping caches. The FPPM Model consistently outperforms the Unrestricted Model for the benchmarks with linearly recursive predicates (i.e., `nrev30` and `vadd16`). Since the

Unrestricted Model allows parallel tasks to create other parallel tasks, it exposes more parallelism in non-linearly recursive predicates than the FPPM Model. Nevertheless, the FPPM model performs better than the Unrestricted Model on some benchmarks (i.e.,`fib7` and `tak963`) that contain non-linearly recursive predicates.

6.7. Summary

The VLSI-BAM is the fastest sequential Prolog implementation that we are aware of. The sequential FPPM implementation that we use in our evaluation performs better than the VLSI-BAM (primarily because FPPM programs are written by hand). Therefore, the speedups due to parallel execution that FPPM achieves represent the true benefits of parallel execution. With an ideal multi-port memory, FPPM achieves speedups between 1.18 and 5.18 (with a geometric mean of 2.05) for a variety of programs that do not create choicepoints or execute side-effects. Although the benchmarks for which these speedups were measured are small, we believe that they are representative of the deterministic parts of programs that lie between consecutive choicepoints or side-effects.

We investigated FPPM's performance with shared memory implemented using direct-mapped snooping caches for an update as well as an invalidate protocol. All the cache studies were for data caches with a total of 8K words. We also assumed that instruction accesses completed in 1 cycle. Among the cache organizations and protocols studied we found the best performance using the update protocol and a line size of 4 words. With such a cache organization the speedup over sequential execution with a similar cache (but without the cache coherence overhead) ranged from 1.09 to 3.17 (with a geometric mean of 1.80).

The two most expensive specialized resources of FPPM are the special buses and the write-once register sets. We found that the performance penalty for a single bus instead of separate task dispatch and result distribution buses is small. However, if we eliminate both the buses then all shared data must reside in shared memory. This increases the overhead of creating parallel tasks. Consequently, parallel execution achieves smaller speedup and requires bigger tasks to perform better than sequential execution.

The number of write-once register sets limits the number of procedures that have parallel tasks in execution. If the number of write-once register sets is reduced from 8 to 4, then the geometric mean speedup (assuming ideal memory) decreases from 2.05 to 1.65.

FPPM's model of execution allows only the main program to create parallel tasks. This limits the amount of parallelism that can be exploited for non-linearly recursive predicates. The Unrestricted Model is an alternative that allows parallel tasks to create other parallel tasks, but it incurs greater overhead than FPPM's model. In a performance comparison of the two models, we found that the Unrestricted Model performs better than the FPPM Model for some non-linearly recursive predicates, but performs worse than the FPPM model for the linearly recursive predicates and for other non-linearly recursive predicates.

7. Conclusions

7.1. Summary of Research Goals

The potential usefulness of Prolog has motivated several attempts to achieve higher performance than the fastest sequential Prolog system currently available. Several researchers have attempted to do this by exploiting parallelism in Prolog. Most of these efforts have concentrated on coarse grain forms of parallelism such as AND-parallelism and OR-parallelism. These forms of parallelism provide potential for many parallel tasks. However, AND- and OR-parallel systems require complicated task management and memory management schemes that result in high overhead. Consequently, few practical systems have demonstrated speedups over the most efficient sequential Prolog systems. The goals of this dissertation have been

- to identify forms of parallelism that could be exploited with low overhead memory management and task management schemes,
- to develop a model of parallel execution that exploits flow parallelism efficiently,
- to identify architectural features that are useful in exploiting flow parallelism, and
- to measure the performance of the architecture and examine the tradeoffs in the design.

7.2. Summary of Research Contributions

(1) We identified a form of fine grain parallelism, called flow parallelism, that can be exploited with low overhead memory management and task management schemes. Flow parallelism is exploited when goals that do not create choicepoints nor cause side-effects are executed in parallel. Thus, flow parallelism is a special case of AND-parallelism. General AND-parallel goals are usually of coarser granularity than flow parallel goals because they are allowed to create choicepoints. We showed that unification and complex goal argument creation are special goals (called unification goals) that perform explicit unification. Since unification goals do not create choicepoints nor cause side-effects, unification parallelism is a special case of flow parallelism. Some bookkeeping operations, such as environment allocation and deallocation, can also execute in parallel with low overhead.

(2) We defined a model of parallel execution that exploits flow parallelism while requiring low overhead. The model ensures that deadlock will not occur if tasks busy-wait for dependencies to be satisfied. Therefore, the overhead due to task suspension and context switching is eliminated. In the model, a program consists of a main program, executing on a Main Processor, and parallel tasks created by the main program that execute on Slave Processors. Parallel tasks executing on the Slave Processors are not allowed to create other parallel tasks. Task creation and scheduling for this master-slave model can be implemented with very low overhead. We also described an alternative scheme, called the Unrestricted Scheme, in which parallel tasks are allowed to create other parallel tasks. In order to prevent deadlock without context switching, the Unrestricted Scheme requires free processors to queue for tasks. A task may create a parallel task only if a free processor is available in the processor queue; if no processor is available, then the task is executed sequentially. The Unrestricted Scheme can expose more parallelism than the master-slave model in non-linearly recursive programs, but task creation and scheduling in the Unrestricted Scheme involve greater overhead.

(3) We specified an architecture, called the Flow Parallel Prolog Machine (FPPM) that exploits flow parallelism using the master-slave model. FPPM consists of a Main Processor that executes the main program and a number of Slave Processors that execute parallel tasks. The Main Processor and the Slave Processors access shared memory. The FPPM architecture has additional features to support Prolog execution, data dependency handling between parallel tasks, and fast task creation and termination.

(4) We described an implementation of FPPM in which each processor can be implemented as a VLSI chip and shared memory is implemented using snooping caches. We described implementations for both update and invalidate protocols.

(5) We evaluated the performance of FPPM and investigated the design tradeoffs using measurements from a detailed register transfer level simulator. Specifically, we obtained the following results:

- Sequential Prolog execution on a single processor (the Main Processor) is about as fast as the Berkeley VLSI-BAM [40], currently the fastest sequential Prolog processor.

- Assuming an ideal shared memory (i.e., one cycle memory access for each port), FPPM achieves speedups ranging from 1.18 to 5.18 (with a geometric mean of 2.05) for the benchmarks using up to 7 Slave Processors.

- If shared memory is implemented using 8K word direct-mapped snooping caches (for data) and an update cache coherence protocol, FPPM achieves speedups ranging from 1.09 to 3.17 (with a geometric mean of 1.8) over the sequential execution with the same size data cache (but without the overhead of a cache coherence protocol). The invalidate protocol does not perform as well as the update protocol.

- FPPM cannot speed up programs that create choicepoints frequently and perform very small computations between choicepoints. However, the overhead of parallel execution is small enough that trying to execute one such program, the *queens4* benchmark, results in only 2% lower performance than sequential execution.

- The result distribution bus and the task dispatch bus are not heavily used. Therefore, there is no need to provide separate buses; a single bus can be used for both purposes with no appreciable performance degradation. However, if we eliminate both buses all shared data must reside in shared memory and explicit synchronization is required. This increases the overhead of creating parallel tasks and the speedup over sequential execution is achieved only for large tasks. Reducing the number of register sets from 8 to 4 reduces the speedup (assuming ideal shared memory) over sequential execution from 2.05 to 1.65.

- FPPM's master-slave model of parallel execution exposes only a limited amount of parallelism in non-linearly recursive predicates. The Unrestricted Scheme does not suffer from this limitation, but the overhead for task creation and scheduling is greater. As expected, we found that FPPM performs better than the Unrestricted Scheme for the linearly recursive programs. We found that FPPM performs better than the Unrestricted Scheme even for some non-linearly recursive predicates. Although our benchmark suite has more non-linearly recursive programs than linearly recursive programs, the mean performance of FPPM is about 25% better than the Unrestricted Scheme.

7.3. Directions for Future Research

The model of parallel execution proposed in this dissertation places restrictions on FPPM tasks that reduce the overhead for parallel execution; unfortunately, the also limit the available parallelism. The two restrictions that have the most significant impact on the available parallelism are: (1) Parallel tasks cannot create choicepoints or cause side-effects and (2) Parallel tasks cannot create other parallel tasks. Several opportunities for future research exist in both easing these restrictions and finding other ways to exploit parallelism that FPPM does not exploit. We describe some of these below.

Restriction (1) makes it important for Prolog compilers to eliminate choicepoint creation wherever possible. With good mode analysis, compilers should generate code to create choicepoints only for those predicates that require a non-deterministic search strategy. FPPM cannot directly exploit OR-parallelism that is inherent in such predicates. One avenue of research is to exploit OR-parallelism using a network of FPPMs without substantially increasing the overhead of flow parallel execution within each deterministic branch of the search. However, the creation of such OR-parallel processes will most probably require much larger overhead than creation of FPPM tasks. Consequently, OR-parallelism cannot be exploited effectively in this way if many OR-branches in the search tree are small, as they often are near the leaves of a search tree. Determining the size of an OR-parallel branch is a hard problem. The alternative, an unpleasant one, is to require the user to annotate which branches of the search tree should be explored in parallel.

Another alternative is to ease restriction (1) so that parallel tasks are allowed to create choicepoints on their local stacks to permit backtracking within the task under the following circumstances:

(a) No choicepoints remain on the local task when it completes. This is because no context is saved for tasks.

(b) During backtracking the task never unbinds shared variables. This is because other tasks could have read the bound variable and must also fail, requiring a global choicepoint rather than a choicepoint local to a task.

(c) If the task has allocated space on the shared heap, this space cannot be reclaimed on backtracking since other tasks could have space allocated above it.

(d) The task must maintain a local trail stack since the global trail stack contains addresses of variables bound by other tasks.

Restriction (2) simplifies scheduling and helps eliminate context switching. However, with this restriction the partitioning of the program into parallel tasks becomes critical in obtaining good speedups. In this dissertation we use a heuristic that relies on the recursive structure of programs to determine which goals to execute in parallel. However, the actual parallelism available depends on the data dependencies between the parallel tasks. A promising research topic is a more sophisticated heuristic that also uses the data dependency information derived from flow analysis. Similar research on data dependency analysis of Lisp programs to create parallel programs was done by Larus [43].

References

1. L. Alkalaj and E. Shapiro, An Architectural Model for a Flat Concurrent Prolog Processor, *Proceedings of the 5th International Conference and Symposium on Logic Programming 2* (1988), 1277-1297.

2. Arvind and K. P. Gostelow, The U Interpreter, *IEEE Computer 15* (Feb 1982), 42-50, IEEE.

3. U. Baron, J. C. Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J. Syre and H. Westphal, The Parallel ECRC Prolog System PEPSys : an Overview and Evaluation of Results, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988, 841-850.

4. J. Beer, Concepts, Design, and Performance Analysis of a Parallel Prolog Machine, *PhD thesis, Technical University*, Berlin, .

5. H. Benker, J. M. Beacco, S. Bescos, M. Dorochevsky, T. Jeffre, A. Pohlmann, J. Noye, B. Poterie, A. Sexton, J. C. Syre, O. Thiebault and G. Waltzlawik, KCM: A Knowledge Crunching Machine, *Conference Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, May 28 - June 1, 1989, 186-194.

6. P Bitar, A Scheme to Preserve Conventional Prolog Order for Side-Effects in a Parallel Prolog System, *Informal private discussion*, 1988.

7. G. Borriello, A. Cherenson, P. B. Danzig and M. Nelson, RISCs or CISCs for Prolog: A Case Study, *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October, 1987.

8. M. Bruynooghe and G. Janssens, An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, *Proceedings of the 5th International Conference and Symposium on Logic Programming 1* (1988), 669-683, MIT Press.

9. R. Carlson, The Bottom-Up Design of a Prolog Architecture, *Technical Report UCB/Computer Science Dpt. 89/536*, Berkeley, May 1989.

10. M. Carlton and P. Van Roy, A Distributed Prolog System with AND-Parallelism, *IEEE Software Volume 5, No. 1* (January, 1988).

11. J. H. Chang, A. M. Despain and D. DeGroot, AND - Parallelism of Logic Programs Based on a Static Data Dependency Analysis, *Digest of Papers, Spring COMPCON 85*, February 25 - 28, 1985, 218 - 226.

12. C. Chen, A. Singhal and Y. N. Patt, PUP: An Architecture to Exploit Parallel Unification in Prolog, *University of California, Berkeley, Computer Science Division Technical Report No. UCB/Computer Science Dpt. 414*, March 1988.

13. W. Citrin, Parallel Unification Scheduling in Prolog, *PhD thesis, University of California, Berkeley*, Berkeley, California, 1988.

14. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

15. A. Colmerauer, H. Kanoui and M. Caneghem, Etude et Realization d'un System Prolog, *Groupe de Recherche en Intelligence Artificielle*, 1979.

16. J. S. Conery, The AND/OR Model for Parallel Interpretation of Logic Programs, *PhD thesis, Dept. of Information and Computer Science, University of California, Irvine*, 1983

17. P. Cousot and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *4th ACM Symposium on Principles of Programming Languages*, January 1977, 238-252.

18. J. A. Crammond, Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors, *Ph.D. Thesis, Department of Computer Science, Herriot-Watt University*, Edinburgh, May 1988.

19. D. DeGroot, Restricted AND-Parallelism, *Proceedings of the Symposium on Logic Programming*, San Francisco, Aug. 31 - Sep. 3, 1987, 80-89.

20. S. K. Debray, Static Analysis of Parallel Logic Programs, *Proceedings of the 5th International Conference and Symposium on Logic Programming 1* (1988), 711-732, MIT

Press.

21. J. B. Dennis, Data Flow Supercomputers, *IEEE Computer*, Nov 1980, 48-56.

22. T. P. Dobry, Y. N. Patt and A. M. Despain, Design Decisions Influencing the Microarchitecture for a Prolog Machine, *Proceedings of the 17 Annual Workshop on Microprogramming (MICRO-17)*, Oct 1984.

23. T. P. Dobry, A. M. Despain and Y. N. Patt, Performance Studies of a Prolog Machine Architecture, *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985.

24. T. Dobry, A Coprocessor for AI, Lisp, Prolog and Databases, *COMPCON*, Spring 1987.

25. T. Dobry, A High Performance Architecture for Prolog, *PhD thesis, University of California, Berkeley (also available from Kluwer Academic Publisgers, 1990)*, Berkeley, California, 1987.

26. C. Dwork, P. C. Kanellakis and J. C. Mitchell, On the Sequential Nature of Unification, *The Journal of Logic Programming 1* (June 1984), 35-50.

27. E. S. Editor, *Concurrent Prolog: Collected Papers*, MIT Press, Cambridge, Mass., 1987.

28. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.

29. B. S. Fagin, A Parallel Execution Model for Prolog, *PhD thesis, Computer Science Division, Univ. of California, Berkeley*, November, 1987. Available as Tech. Report UCB/Computer Science Dpt./87/380.

30. I. Foster, Parallel Implementation of Parlog, *Proceedings of the International Conference on Parallel Processing*, 1988.

31. I. Foster and S. Taylor, Strand: A Practical Parallel Programming Tool, *Proceedings of the North American Conference on Logic Programming 1* (1989), 497-512.

32. D. D. Gajski, D. A. Padua, D. J. Kuck and R. H. Kuhn, A Second Opinion on Data Flow Machines and Languages, *Computer 15*, 2 (Feb. 1982), 15-25.

33. D. Gajski, D. Kuck, D. Lawrie and A. Sameh, Cedar - A Large Scale Multiprocessor, *Proceedings of the International Conference on Parallel Processing*, 1983, 524-529.

34. J. Gee, S. W. Melvin and Y. N. Patt, Advantages of Implementing PROLOG by Microprogramming a Host General Purpose Computer, *Proceedings of the Fourth International Conference on Logic Programming 1* (May 1987), 1-20, MIT Press.

35. J. R. Goodman, Using Cache Memory to Reduce Processor Memory Traffic, *Proceedings of the Tenth Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 5-7, 1983.

36. S. Gregory, *Parallel Logic Programming in PARLOG*, 1987.

37. A. Harsat and R. Ginosar, Carmel 2: A Second Generation VLSI Architecture for Flat Concurrent Prolog, *Proceedings of the International Conference on Fifth Genereation Computer Systems*, Tokyo, 1988, 962-969.

38. R. Hasegawa and M. Amamiya, Parallel Execution of Logic Programs based on Dataflow Concept, *Proceedings of the International Conference on Fifth Generation Computer Systems, 1984*, 1984, 507-516.

39. M. V. Hermenegildo, An Abstract Machine for the Restricted AND-Parallelism of Logic Programs, *Third International Conference on Logic Programming*, July, 1986, 25-39.

40. B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. R. Bush and A. M. Despain, Fast Prolog with an Extended General Purpose Architecture, *17th International Symposium on Computer Architecture*, Sea [illegible], May 1990.

41. N. Ichiyoshi, T. Miyazaki and K. Taki, A Distributed Implementation of Flat GHC on the Multi-PSI, *Proceedings of the 4th International Conference on Logic Programming*, 1987, 257-275.

42. N. Ito, H. Shimizu, M. Kishi, E. Kuno and K. Rokusawa, Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog, *New Generation Computing 3* (1985), 15-41, OHMSHA, LTD and Springer-Verlag.

43. J. R. Larus, Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors, *PhD. Thesis, also Technical Report UCB/Computer Science Dpt. 89/502*, Berkeley, May 1989.

44. Y. Lin and V. Kumar, Performance of AND-Parallel Execution of Logic Programs on a Shared-Memory Multiprocessor, *Proceedings of the International Conference on Fifth Generation Computer Systems 3* (Nov 28 - Dec 2, 1988), 851 - 860, Institute for New Generation Computer Technology.

45. E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski and B. Hausman, The Aurora Or-Parallel Prolog System, *Proceeedings of the Symposium on Logic Programming*, 1988.

46. J. W. Mills, A High Performance LOW RISC Machine for Logic Programming, *Addendum to Proceedings, Third Symposium on Logic Programming*, Salt Lake City, Utah, Sept. 21 - 25, 1986.

47. M. Morioka, S. Yamaguchi and T. Bandoh, Evaluation of Memory System for Integrated Prolog Processor IPP, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, May 28- June 1, 1989, 203-210.

48. H. Mulder and E. Tick, A Performance Comparison Between PLM and an MC68020 Prolog Processor, *Technical Note no. CSL-86-302, Computer Systems Laboratory, Stanford University*, Stanford, California, September, 1986.

49. H. Nakashima and K. Nakajima, Hardware Architecture of the Sequential Inference Machine: PSI-II, *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, 1987, 104 - 113.

50. Y. N. Patt, W. Hwu and M. C. Shebanow, HPS, A New Microarchitecture: Rationale and Introduction, *Proceedings of the 18th International Microprogramming Workshop*, Asilomar, California, December, 1985.

51. Y. N. Patt and C. Chen, A Comparison Between the PLM and the MC 68020 as Prolog Processors, *Report No. UCB/Computer Science Dpt. 87/397*, Berkeley, Jan 1988.

52. J. L. Peterson and A. Silberschatz, *Opera:. .g System Concepts*, Addison-Wesley Publishing Company, 1985.

53. Quintus Computer Systems Inc., Quintus Prolog Reference Manual, February, 1987.

54. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle, The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, Tradeoffs, *IEEE Computer*, January 1989.

55. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, 1989. (Ph.D. Dissertation, Stanford University).

56. E. Shapiro, The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys 21 No. 3* (September 1989), 412-510.

57. W. Shu, B. Ramkumar and L. V. Kale, Implementation and Performance of a Parallel Prolog Interpreter, *Technical Report no. UIUCDCS-R-88-1480*, December, 1988.

58. A. Singhal and Y. N. Patt, Unification Parallelism: How much can we exploit?, *Proceedings of the North American Conference on Logic Programming*, Cleveland, Oct 16-20, 1989.

59. A. Singhal and Y. N. Patt, A High Performance Prolog Processor with Multiple Function Units, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, May 28 - June 1, 1989, 195-202.

60. V. P. Srini, J. V. Tam, T. M. Nguyen, A. M. Despain, M. Moll and D. Ellsworth, A CMOS Chip for Prolog, *Proceedings of the International Conference on Computer Design*, October, 1987.

61. J. Syre and H. Westphal, A Review of Parallel Models for Logic Programming Languages, *Technical Report CA-07, European Computer Industry Research Centre, GmbH, Arabellastr, 17, D-8000 Muenchen 81, West Germany*, 10 June 1985.

62. S. Taylor, E. Av-Ron and E. Shapiro, A Parallel Implementation of Flat Concurrent Prolog, *Jornal of Parallel Programming 15, No. 3* (1987), 245-275.

63. A. Taylor, LIPS on a MIPS: Results from a Prolog Compiler for a RISC, *International Conference on Logic Programming*, Jerusalem, June 1990.

64. E. Tick, Studies in Prolog Architectures, *Phd thesis (also Technical Report No. CSL-Tech. Rep.-87-329, Computer Systems Laboratory, Stanford University)*, Stanford, California, June, 1987.

65. R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development 11* (1967).

66. H. Touati and A. Despain, An Empirical Study of the Warren Abstract Machine, *Proceedings of the Symposium on Logic Programming*, San Francisco, California, Aug 31- Sep 4, 1987, 114-124.

67. P. Van Roy, An Intermediate Language to Support Prolog's Unification, *Proceedings of the North American Conference on Logic Programming 2* (October 1989), 1136-1148, MIT Press.

68. P. Van Roy and A. M. Despain, The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, *North American Conference on Logic Programming*, 1990.

69. P. Van Roy, Can logic programming execute as efficiently as imperative programming?, *PhD Thesis (in preparation)*, Berkeley, .

70. J. S. Vitter and R. A. Simmons, New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems for P, *IEEE Transactions on Computers 35 Number 5* (May 1986).

71. D. H. D. Warren, An Abstract Prolog Instruction Set, *Technical Report 309, Artificial Intelligence Center, SRI International*, 1983.

72. H. Westphal and P. Robert, The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism, *Proceedings of the Symposium on Logic Programming*, San Francisco, Aug. 31 - Sep. 4, 1987, 636-648.

73. H. Yasuura, On Parallel Computational Complexity of Unification, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

74. Zycad Corporation, N.2. User Manuals.

Appendix 1: Detailed Measurements

Bench-mark	Sequent. (cycles)	Number of Slave Processors						
		1	2	3	4	5	6	7
Memory Access Time = 1 cycle								
fib7	754	0.82	1.41	1.70	1.70	1.70	1.70	1.70
hanoi8	8179	0.93	1.82	1.83	1.83	1.83	1.83	1.83
nrev30	4801	0.70	1.22	1.61	1.90	2.11	2.28	2.38
qsort50	5901	0.84	1.18	1.18	1.18	1.18	1.18	1.18
qsort50r	5901	0.84	1.38	1.63	1.81	1.81	1.81	1.81
tak963	3374	1.00	1.83	1.83	1.83	1.83	1.83	1.83
vadd16	7585	0.89	1.72	2.56	3.27	3.94	4.70	5.18
arith. mean	-	0.86	1.51	1.76	1.93	2.06	2.19	2.27
geom. mean	-	0.86	1.49	1.72	1.85	1.93	2.01	2.05
Memory Access Time = 2 Cycles								
fib7	1136	0.92	1.63	1.92	1.92	1.92	1.92	1.92
hanoi8	14045	0.97	1.89	1.89	1.89	1.89	1.89	1.89
nrev30	7547	0.80	1.40	1.86	2.21	2.49	2.69	2.85
qsort50	8497	0.89	1.24	1.24	1.24	1.24	1.24	1.24
qsort50r	8497	0.89	1.47	1.72	1.90	1.90	1.90	1.90
tak963	4615	1.00	1.84	1.84	1.84	1.84	1.84	1.84
vadd16	10721	0.95	1.85	2.72	3.51	4.18	5.04	5.33
arith. mean	-	0.92	1.62	1.88	2.07	2.21	2.36	2.45
geom. mean	-	0.92	1.60	1.84	1.99	2.07	2.15	2.20
Memory Access Time = 3 Cycles								
fib7	1517	0.98	1.76	2.10	2.10	2.10	2.10	2.10
hanoi8	19911	0.98	1.92	1.92	1.92	1.92	1.92	1.92
nrev30	10293	0.85	1.51	2.01	2.39	2.69	2.92	3.09
qsort50	11093	0.92	1.27	1.27	1.27	1.27	1.27	1.27
qsort50r	11093	0.92	1.52	1.77	1.95	1.95	1.95	1.95
tak963	5856	1.00	1.85	1.85	1.85	1.85	1.85	1.85
vadd16	13857	0.99	1.92	2.82	3.66	4.33	5.23	5.75
arith. mean	-	0.95	1.68	1.96	2.16	2.30	2.46	2.56
geom. mean	-	0.95	1.66	1.91	2.07	2.15	2.24	2.29
Memory Access Time = 4 Cycles								
fib7	1898	1.01	1.82	2.14	2.14	2.14	2.14	2.14
hanoi8	25777	0.99	1.93	1.93	1.93	1.93	1.93	1.93
nrev30	13039	0.89	1.58	2.11	2.53	2.86	3.13	3.33
qsort50	13689	0.94	1.29	1.29	1.29	1.29	1.29	1.29
qsort50r	13689	0.94	1.56	1.80	1.98	1.98	1.98	1.98
tak963	7079	1.00	1.85	1.85	1.85	1.85	1.85	1.85
vadd16	16993	1.01	1.97	2.89	3.76	4.43	5.38	5.89
arith. mean	-	0.97	1.71	2.00	2.21	2.36	2.53	2.63
geom. mean	-	0.97	1.70	1.95	2.11	2.20	2.29	2.34

Table A1.1: Speedup relative to sequential execution for memory access times varying from 1 to 4. FPPM implementation with 8 write-once register sets is assumed.

Bench-mark	Sequent. (cycles)	Number of Slave Processors 1	2	3	4	5	6	7
Update Protocol, 2 words/line, 4096 lines (dcacheu2)								
fib7	1167	1.02	1.83	2.27	2.30	2.32	2.31	2.31
hanoi8	8615	0.83	1.53	1.53	1.53	1.53	1.53	1.53
nrev30	5913	0.74	0.96	1.23	1.44	1.56	1.54	1.51
qsort50	6021	0.76	1.05	1.05	1.05	1.05	1.05	1.05
qsort50r	6021	0.77	1.21	1.36	1.54	1.55	1.55	1.55
tak963	3374	0.98	1.79	1.79	1.79	1.79	1.79	1.79
vadd16	7753	0.79	1.28	1.79	1.79	2.55	2.90	3.01
arith mean	-	0.84	1.38	1.57	1.63	1.76	1.81	1.82
geom mean	-	0.84	1.34	1.53	1.59	1.70	1.73	1.73
Update Protocol, 4 words/line, 2048 lines (dcacheu4)								
fib7	1219	1.01	1.93	2.35	2.40	2.42	2.42	2.42
hanoi8	8607	0.88	1.63	1.63	1.63	1.62	1.63	1.62
nrev30	5635	0.75	0.98	1.26	1.43	1.60	1.63	1.55
qsort50	6021	0.81	1.10	1.09	1.09	1.09	1.09	1.09
qsort50r	6021	0.81	1.27	1.42	1.60	1.59	1.60	1.59
tak963	3374	0.99	1.81	1.81	1.81	1.81	1.81	1.81
vadd16	7775	0.84	1.34	1.92	2.50	2.65	3.14	3.17
arith mean	-	0.87	1.44	1.64	1.78	1.83	1.90	1.89
geom mean	-	0.86	1.40	1.59	1.72	1.76	1.81	1.80
Update Protocol, 8 words/line, 1024 lines (dcacheu8)								
fib7	1533	1.12	2.25	2.68	3.01	2.98	2.98	2.98
hanoi8	8587	0.89	1.61	1.65	1.64	1.64	1.64	1.63
nrev30	5633	0.78	0.90	1.27	1.37	1.47	1.48	1.39
qsort50	6019	0.83	1.10	1.09	1.08	1.08	1.08	1.08
qsort50r	6019	0.82	1.27	1.40	1.56	1.56	1.57	1.55
tak963	3374	0.99	1.82	1.82	1.82	1.82	1.82	1.82
vadd16	7817	0.86	1.44	1.89	2.38	2.50	2.87	3.02
arith mean	-	0.90	1.48	1.69	1.84	1.86	1.92	1.92
geom mean	-	0.89	1.42	1.62	1.74	1.77	1.81	1.80

Table A1.2: Performance of FPPM for various data cache organizations with the update coherence protocol relative to sequential execution with the same cache organization. The size of each cache is 8K words. For sequential execution we simulate a cache of the same size and organization but without the overhead of a coherence protocol. A FPPM implementation with 8 register sets is assumed.

Bench-mark	Sequent. (cycles)	Number of Slave Processors 1	2	3	4	5	6	7
Invalidate Protocol, 2 words/line, 4096 lines (dcachei2)								
fib7	1167	1.01	1.74	2.08	2.17	2.16	2.16	2.16
hanoi8	8615	0.83	1.53	1.53	1.53	1.53	1.53	1.53
nrev30	5913	0.74	0.97	1.24	1.42	1.34	1.32	1.30
qsort50	6021	0.76	1.06	1.04	1.04	1.04	1.04	1.04
qsort50r	6021	0.77	1.21	1.35	1.54	1.54	1.54	1.53
tak963	3374	0.98	1.79	1.79	1.79	1.79	1.79	1.79
vadd16	7753	0.79	1.26	1.81	2.25	2.54	2.94	2.98
arith mean	-	0.84	1.37	1.55	1.68	1.70	1.76	1.76
geom mean	-	0.84	1.33	1.51	1.63	1.64	1.67	1.67
Invalidate Protocol, 4 words/line, 2048 lines (dcachei4)								
fib7	1219	1.00	1.80	2.19	2.23	2.22	2.21	2.21
hanoi8	8607	0.88	1.63	1.63	1.63	1.62	1.63	1.62
nrev30	5635	0.75	0.98	1.24	1.43	1.40	1.37	1.28
qsort50	6021	0.81	1.09	1.07	1.07	1.07	1.07	1.07
qsort50r	6021	0.81	1.25	1.37	1.54	1.55	1.54	1.53
tak963	3374	0.99	1.81	1.81	1.81	1.81	1.81	1.81
vadd16	7775	0.84	1.31	1.88	2.45	2.64	3.10	3.12
arith mean	-	0.87	1.41	1.60	1.74	1.76	1.82	1.81
geom mean	-	0.86	1.37	1.55	1.68	1.69	1.72	1.71
Invalidate Protocol, 8 words/line, 1024 lines (dcachei8)								
fib7	1533	1.11	2.06	2.37	2.68	2.62	2.63	2.63
hanoi8	8587	0.89	1.62	1.65	1.64	1.64	1.64	1.63
nrev30	5633	0.78	0.97	1.16	1.22	1.13	1.07	1.06
qsort50	6019	0.83	1.06	1.01	0.99	0.99	1.00	0.99
qsort50r	6019	0.82	1.21	1.28	1.44	1.37	1.40	1.40
tak963	3374	0.99	1.82	1.82	1.82	1.82	1.82	1.82
vadd16	7817	0.86	1.31	1.87	2.27	2.48	2.74	2.87
arith mean	-	0.90	1.44	1.59	1.72	1.72	1.76	1.77
geom mean	-	0.89	1.39	1.53	1.64	1.62	1.64	1.65

Table A1.3: Performance of FPPM for various data cache organizationswith the invalidate coherence protocol relative to sequential execution with the same cache organization. The size of each cache is 8K words. For sequential execution we simulate a cache of the same size and organization but without the overhead of a coherence protocol. A FPPM implementation with 8 register sets is assumed.

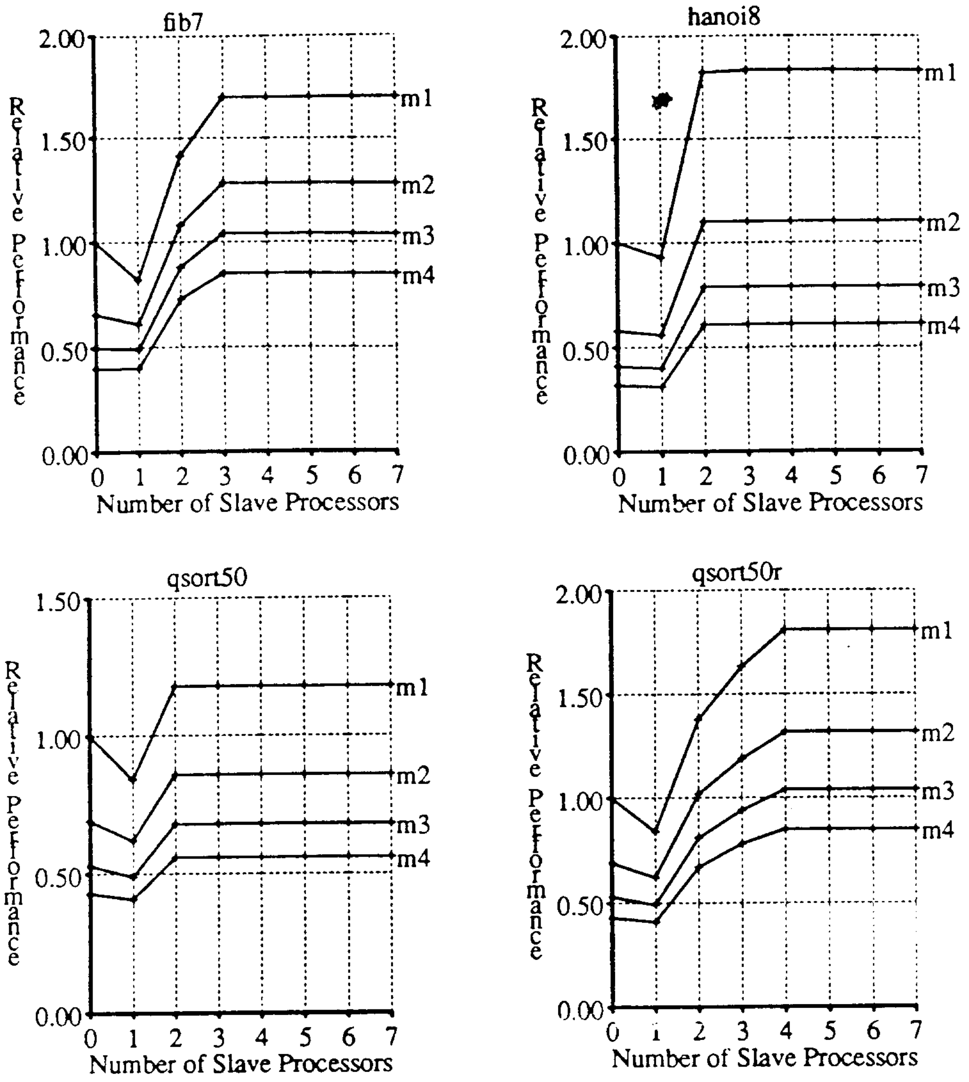
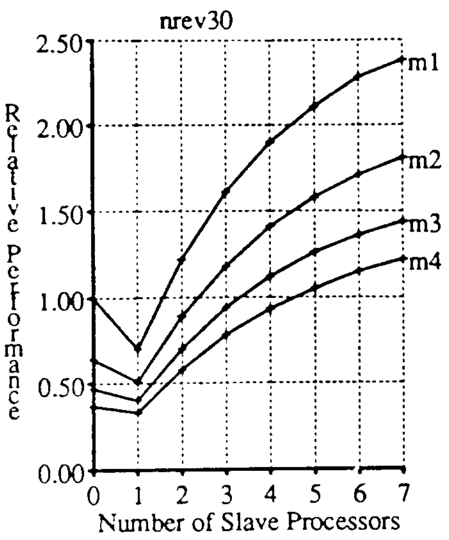
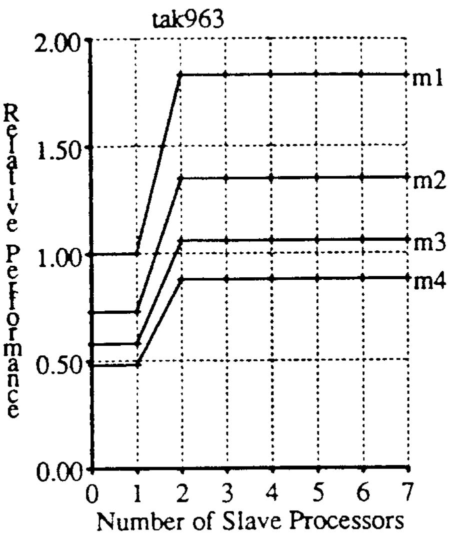


Figure A1.1(a): Effect of memory access time on the performance of FPPM relative to sequential execution with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors.





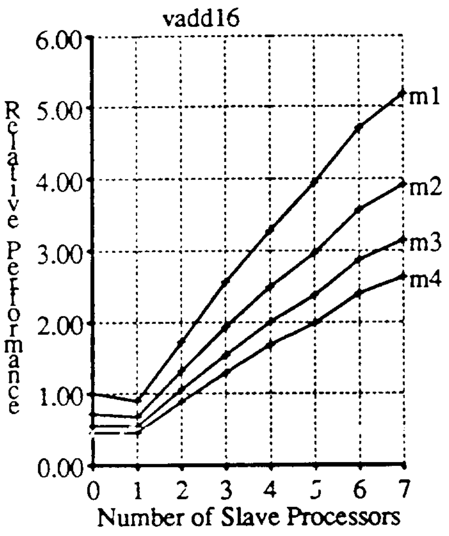


Figure A1.1(b): Effect of memory access time on the performance of FPPM relative to sequential execution with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors.

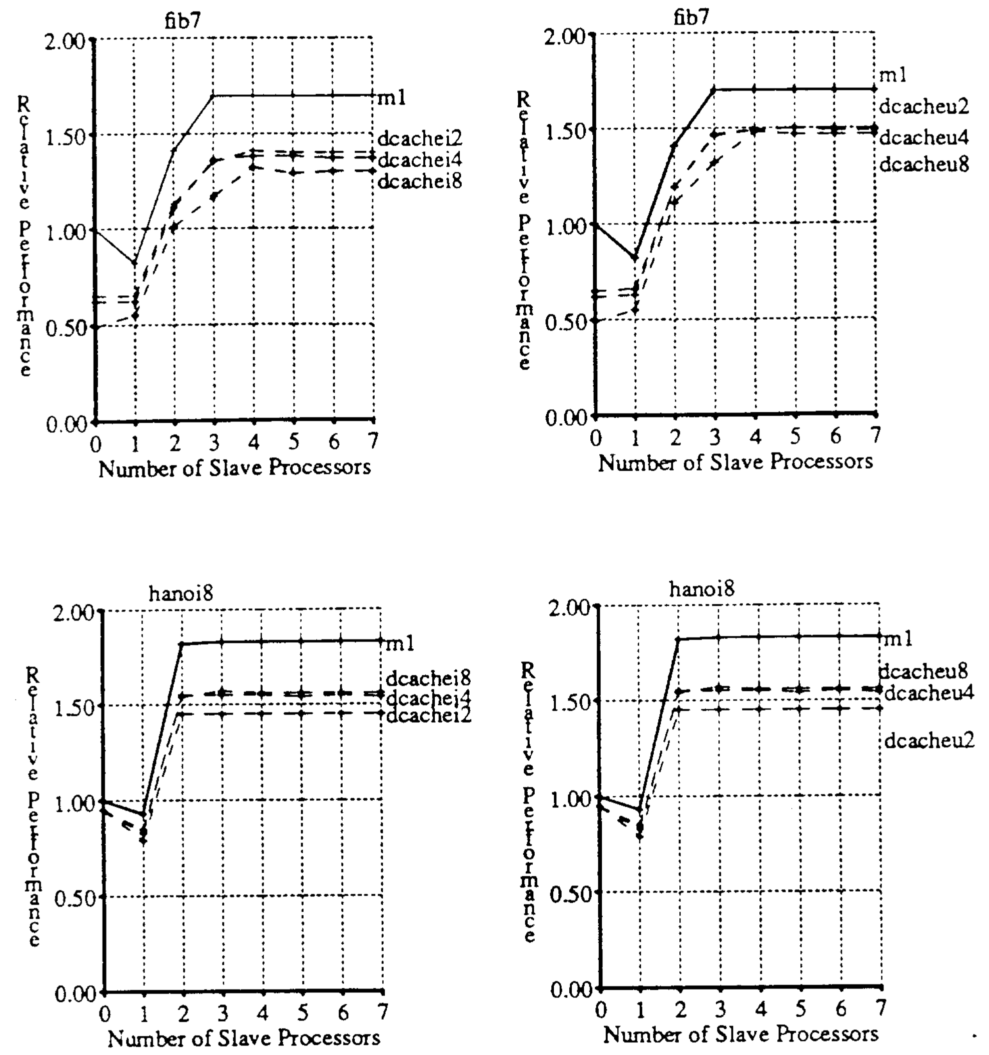


Figure A1.2 (a): Performance of FPPM with data caches relative to ideal sequential performance with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. No coherence protocol is used for the sequential case. A FPPM implementation with 8 register sets is assumed. The lines titled "dcache*ps*" are for data caches where *p* is the protocol (i is invalidate, u is update) and *s* is the line size. The m1 line, representing performance with ideal 1-cycle multi-port memory, is drawn for reference.

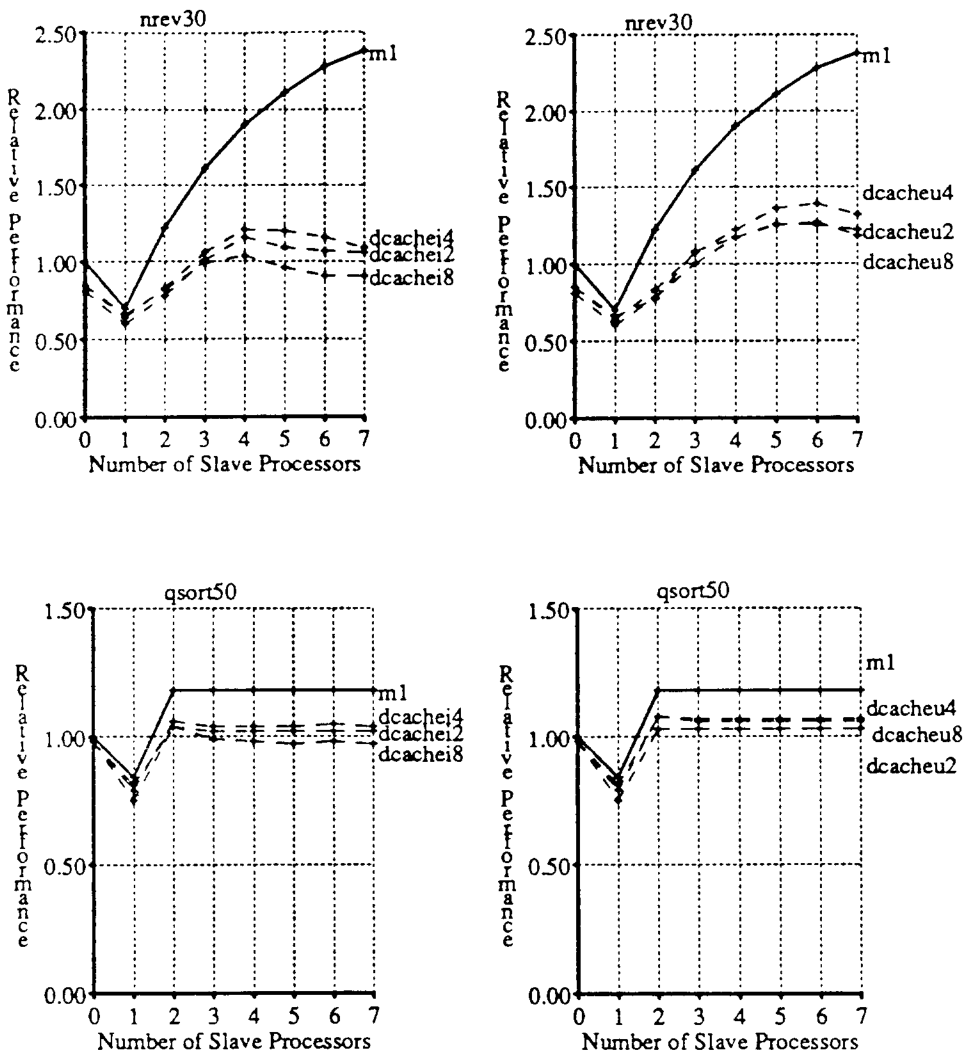


Figure A1.2 (b): Performance of FPPM with data caches relative to ideal sequential performance with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. No coherence protocol is used for the sequential case. A FPPM implementation with 8 register sets is assumed. The lines titled "dcache*ps*" are for data caches where *p* is the protocol (i is invalidate, u is update) and *s* is the line size. The ml line, representing performance with ideal 1-cycle multi-port memory, is drawn for reference.

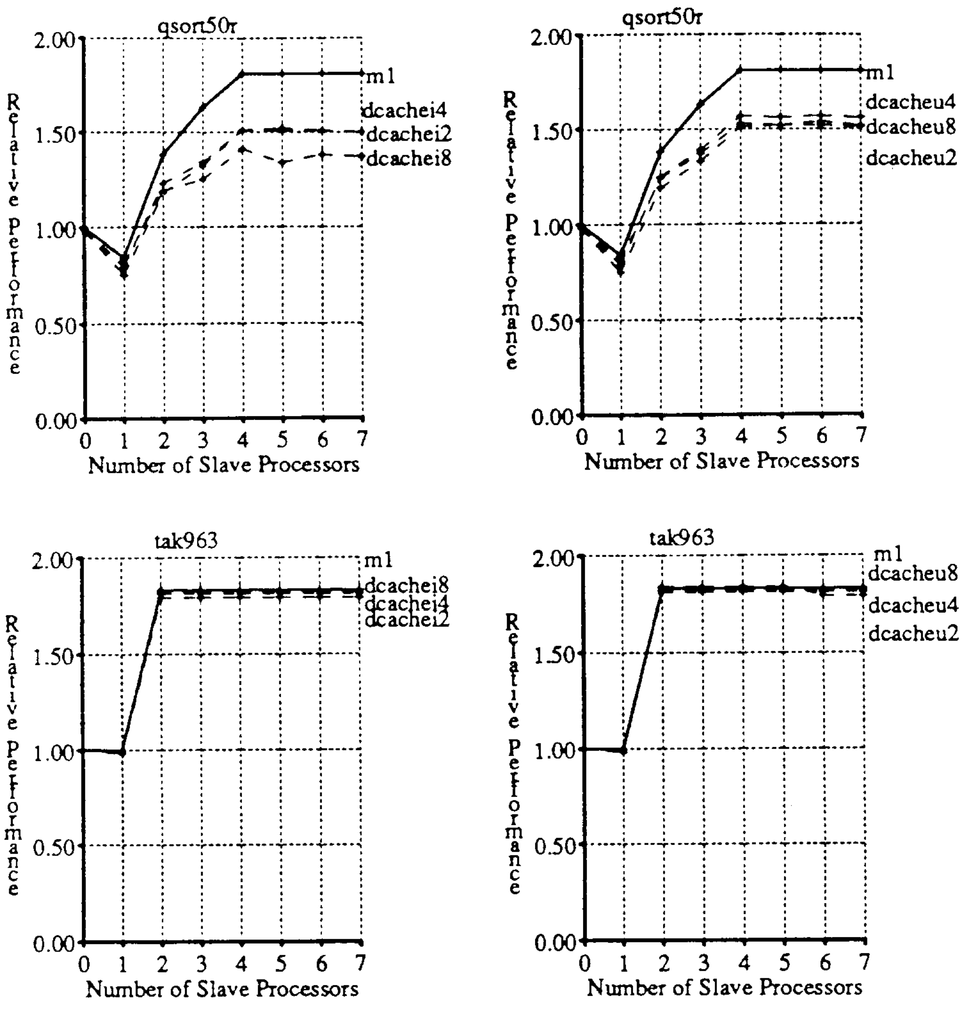
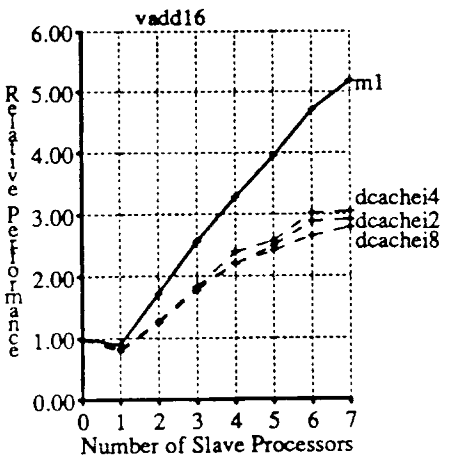


Figure A1.2 (c): Performance of FPPM with data caches relative to ideal sequential performance with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. No coherence protocol is used for the sequential case. A FPPM implementation with 8 register sets is assumed. The lines titled "dcache*ps*" are for data caches where *p* is the protocol (i is invalidate, u is update) and *s* is the line size. The m1 line, representing performance with ideal 1-cycle multi-port memory, is drawn for reference.



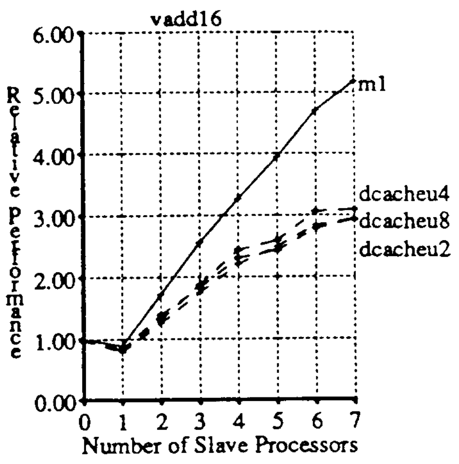


Figure A1.2 (d): Performance of FPPM with data caches relative to ideal sequential performance with 1 cycle memory access. On the X-axis sequential program execution (using the Main Processor alone) is represented by data points for 0 Slave Processors. No coherence protocol is used for the sequential case. A FPPM implementation with 8 register sets is assumed. The lines titled "dcache*ps*" are for data caches where *p* is the protocol (i is invalidate, u is update) and *s* is the line size. The ml line, representing performance with ideal 1-cycle multi-port memory, is drawn for reference.

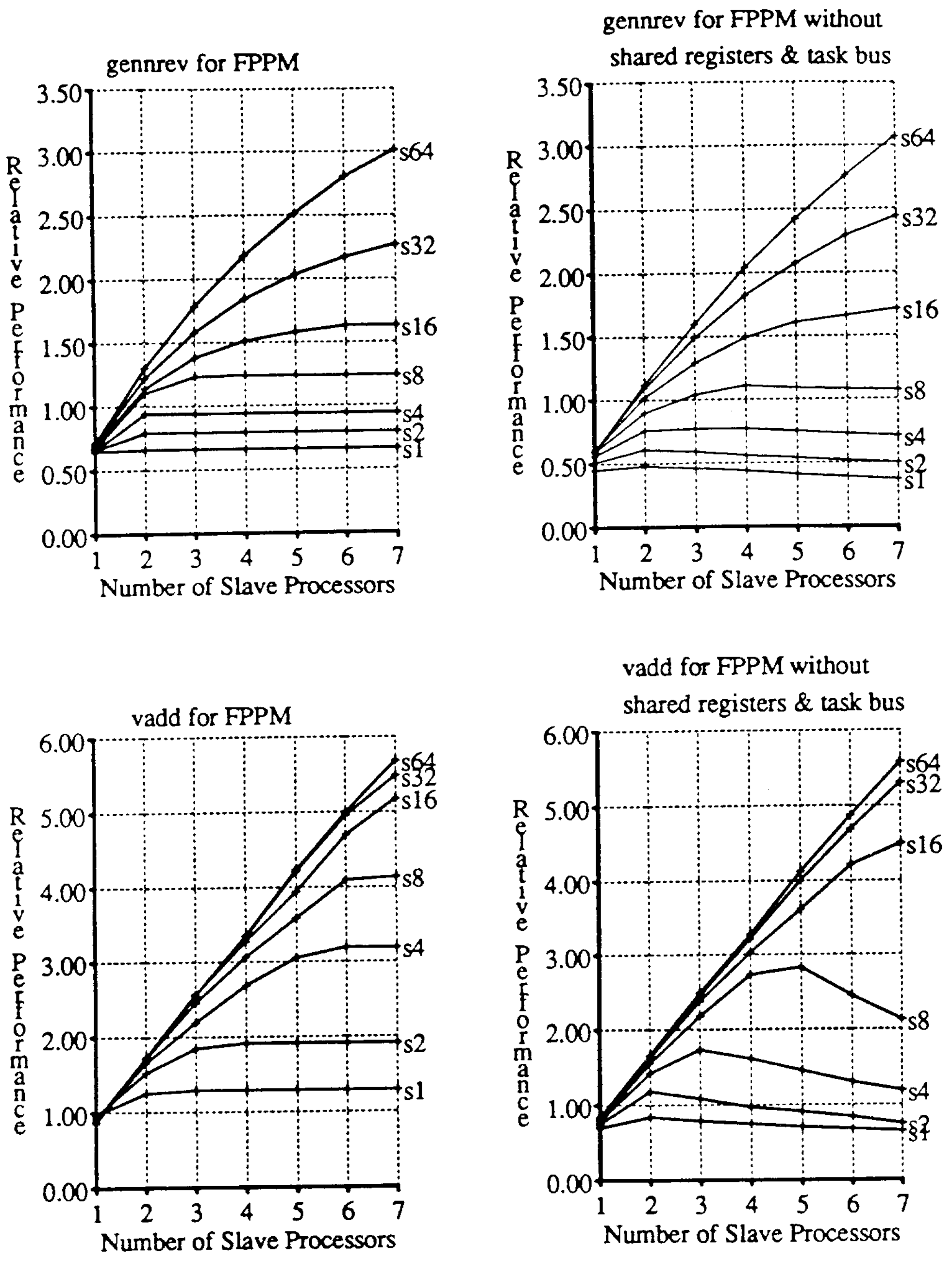


Figure A1.3 (a): Performance of FPPM with ideal shared memory relative to sequential performance for the gennrev*n* and vadd*n* benchmarks for various values of *n*. Speedup due to parallel execution is greater for larger values of *n*. The speedups are lower for FPPM without the shared registers and task bus, especially for small values of *n*.

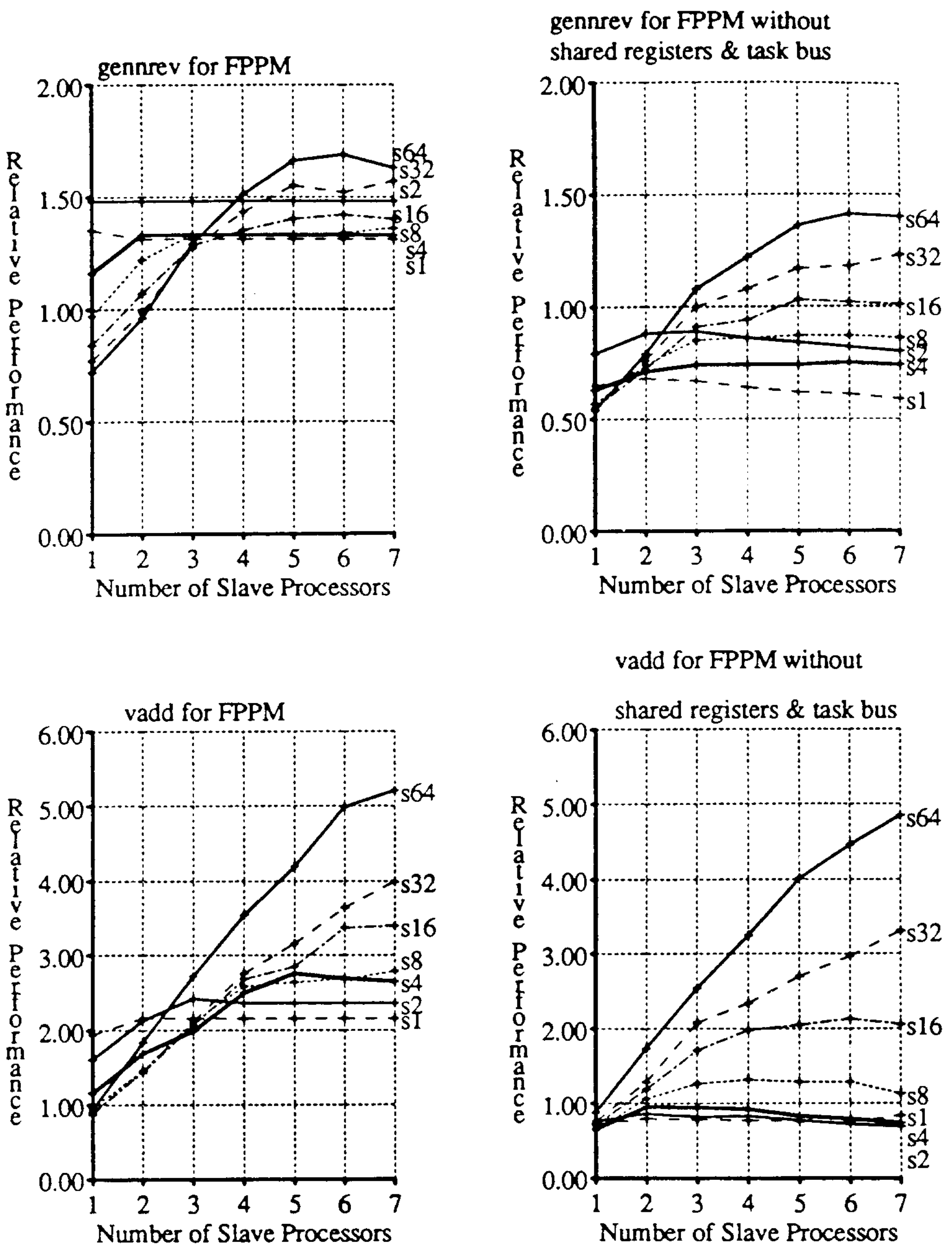


Figure A1.3 (b): Performance of FPPM with shared memory implemented using 8K word, direct-mapped snooping-caches with 4 words/line and an update protocol relative to sequential performance for the gennrev*n* and vadd*n* benchmarks for various values of *n*. The difference in performance between FPPM with and without the shared registers and task bus is greater than in figure A1.3(a).

Appendix 2: Benchmarks and FPPM Code

fib7

```
main:-
    fib(7,N).

fib(0,1).
fib(1,1).
fib(N,F):-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1,F1),
    fib(N2,F2),
    F is F1 + F2.
```

hanoi8

```
main :- han(8,1,2,3,X,[]).

han(0,_,_,_,X,X).
han(N,A,B,C,H1,R) :-
    N > 0,
    N1 is N - 1,
    han(N1,A,C,B,H1,[move(A,B)|H2]),
    han(N1,C,B,A,H2,R).
```

nrev30

```
main :- nrev([1,2,3,4,5,6,7,8,9,10,
    11,12,13,14,15,16,17,18,19,20,
    21,22,23,24,25,26,27,28,29,30], L).

nrev([],[]).
nrev([X|L1],L3):-
    nrev(L1,L2),
    concat(L2,[X],L3).

concat([],X,X).
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3).
```

qsort50

```
main :-
    qsort([27,74,17,33,94,18,46,83,65,2,
    32,53,28,85,99,47,28,82,6,11,
    55,29,39,81,90,37,10,0,66,51,
    7,21,85,27,31,63,75,4,95,99,
    11,28,61,74,18,92,40,53,59,8], X,  []).

qsort([X|L],R,R0) :-
    partition(L,X,L1,L2),
    qsort(L2,R1,R0),
    qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
    X<Y,
    partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    X >= Y,
    partition(L,Y,L1,L2).
partition([],_,[],[]).
```

qsort50r

```
main :-
    qsort([74,27,17,33,94,18,46,83,65,2,
    32,53,28,85,99,47,28,82,6,11,
    55,29,39,81,90,37,10,0,66,51,
    7,21,85,27,31,63,75,4,95,99,
    11,28,61,74,18,92,40,53,59,8], X,  []).

qsort([X|L],R,R0) :-
    partition(L,X,L1,L2),
    qsort(L2,R1,R0),
    qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
    X<Y,
    partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    X >= Y,
    partition(L,Y,L1,L2).
partition([],_,[],[]).
```

tak963

```
main :- tak(9,6,3,T).

tak(X,Y,Z,A) :-
        X =< Y,
        Z = A.
tak(X,Y,Z,A) :-
    X > Y,
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).
```

vadd*n*

```
main :-
    createmat(n,n,0,1,M),
    createvec(n,0,2,V,_),
    matadd(M,V,M1),
    print(M1).

vadd([],[],[]).
vadd([X1|R1],[X2|R2],[X3|R3]):-
    X3 is X1 + X2,
    vadd(R1,R2,R3).

matadd([],_,[]).
matadd([V1|R1],V2,[V3|R3]) :-
    vadd(V1,V2,V3),
    matadd(R1,V2,R3).

createvec(0,_,_,[]).
createvec(Size,Start,Step,[Start|Rest]) :-
    Size > 0,
    NSize is Size - 1,
    NStart is Start + Step,
    createvec(NSize,NStart,Step,Rest).

createmat(0, _, _, _,[]).
createmat(Numvec, Vecsize, Start, Step, [V|R]) :-
    Numvec > 0,
    createvec(Vecsize,Start,Step,V),
    NStart is Start + n,
    NNumvec is Numvec - 1,
    createmat(NNumvec,Vecsize,NStart,Step,R).
```

gennrev*n*

```
main :-
    gennrev(n,X).

gennrev(N,X) :-
    range(1,N,L),
    nrev(L,X).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]):-
    M < N,
    M1 is M + 1,
    range(M1,N,Ns).
```

```
nrev([],[]).
nrev([X|L1],L) :- nrev(L1,L2),
    concat(L2,[X],L).

concat([],X,X).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

queens4

```
main:-
    solve(4,X).

solve(N,Qs):-
    range(1,N,Ns),
    queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(Unplaced, Safe, Qs):-
    select(Q, Unplaced, NewUnplaced),
    not_threat(Q,Safe),
    queens(NewUnplaced,[Q|Safe],Qs).

not_threat(Q,Safe):-
    safe(Q,1,Safe).

safe(_,_,[]).
safe(Q,N,[Y|Ys]):-
    Q =\= (Y+N),
    Q =\= (Y-N),
    N1 is N + 1,
    safe(Q,N1,Ys).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-
    select(X,Ys,Zs).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]):-
    M < N,
    M1 is M + 1,
    range(M1,N,Ns).
```

Instruction Definitions and Macros

```
/* FILE: fppm.format */
/* define instruction format for the FPPM processors */

instr inst  [1,1] <32> $

format
    lbit   =    inst[0]<31:31>,       % lock bit %
    ulbit  =    inst[0]<30:30>,       % unlock bit %
    opcode      =    inst[0]<29:24>,
        scode =     inst[0]<23:20>,          % special code for arith,cond%
        itag  =     inst[0]<23:20>,
        off   =     inst[0]<19:12>,
        imm   =     inst[0]<23:12>,
        lab   =     inst[0]<19:6>,
        longlab     =     inst[0]<23:0>,

        reg2t =     inst[0]<17:16>,          % register 2 type %
        reg2  =     inst[0]<15:12>,

        reg0t =     inst[0]<11:10>,
        reg0  =     inst[0]<9:6>,

        reg1t =     inst[0]<5:4>,
        reg1  =     inst[0]<3:0>       $

/* FILE: fppm.m */
/* define instructions */

macro
        ! macros for register types
        IN    =     0     &,    % input set %
        OUT   =     1     &,    % output set %
        G     =     2     &,    % global registers %
        T     =     3     &,    % temprary (local) %

        ! macros for types
        UVAR        =     0x0   &,
        BVAR        =     0x1   &,
        LIST =      0x2   &,
        STRUCT      =     0x3   &,
        INT  =      0x4   &,
        ATOM        =     0x5   &,

        NIL   =     0xf   &,

        ! macros for special shared  registers

        B     =     8     &,
        CP    =     9     &,
        E     =     0xa   &,
        TE    =     0xb   &,
        HP    =     0xc   &,
        TR    =     0xd   &,
        PC    =     0xe   &,
        L     =     0xf   &,

        ! macros for global registers

        TR_REG      =     0xd   &,
        HP_REG      =     0xc   &,
        PC_REG      =     0xe   &,
        FAIL_REG=         0xf   &,
        QREG        =     0x1   &,
        HEADREG =         0x2   &,
        TAILREG  =        0x3   &,
        FRAMEREG =        0xf   &,
        NUMPROC=          0xe   &,
!macros for opcodes

NOP =       0x0  &,
QUIT=       0x1  &,
NEW =       0x2  &,
FAIL =      0x3  &,
DONE        =    0x4  &,
EXEC        =    0x5  &,

WAITEXEC =       0x7  &,

CLEARFAIL =      0x9  &,
OLD =       0xa  &,
CLEARCP =        0xb  &,

ARITH       =    0x10 &,
ARITHOFF =       0x11 &,

MOVETAG          =    0x13 &,
PUSH        =    0x14 &,
PUSHTAG =        0x15 &,

LOAD        =    0x16 &,
STORE       =    0x17 &,

BR   =      0x20 &,
BRIND       =    0x21 &,
BRAL        =    0x22 &,
BRTAGEQ =        0x23 &,
BRTAGNEQ=        0x24 &,
BRCOND  =        0x25 &,
BRNCOND=         0x26 &,
BRCMP       =    0x27 &,
BRNCMP  =        0x28 &,


SBR =       0x30 &,
SBRIND      =    0x31 &,
SBRAL       =    0x32 &,
SBRTAGEQ=        0x33 &,
SBRTAGNEQ=       0x34 &,
SBRCOND =        0x35 &,
SBRNCOND=        0x36 &,
SBRCMP  =        0x37 &,
SBRNCMP =        0x38 &,

! macros for arithmetic codes

ADD =       0x0  &,
SUB =       0x1  &,
NAND        =    0x2  &,
NOR =       0x3  &,
SLL  =      0x4  &,
SRL  =      0x5  &,
MAX =       0x6  &,

! macros for condition codes

GEZ  =      0x0  &,    % >= 0 %
```

```
EQZ =     0x1   &,    % = 0 %
DRF =     0x2   &,    % dereferenced %
OTF =     0x3   &,    % other fail latches %
CPV =     0x4   &,    % valid choicepoints %
VCP =     0x5   &,    % current set is valid choicepoint %

! macros for instructions

nop   = opcode = NOP  $      &,
quit  = opcode = QUIT $      &,
new   = opcode = NEW  $      &,
old   = opcode = OLD  $      &,
fail  = opcode = FAIL $      &,
done  = opcode = DONE        $      &,
waitexec = opcode = WAITEXEC    $      &,

clearfail = opcode = CLEARFAIL   $      &,
clearcp = opcode = CLEARCP$      &,
exec(l)       = opcode = EXEC;
        longlab = l  $      &,

arith(c,r1t,r1,r0t,r0,r2t,r2) =            ! r1 <- r0 c r2
        opcode = ARITH;
        scode = c;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        reg2t = r2t;
        reg2 = r2    $      &,

arithoff(c,r1t,r1,r0t,r0,v) =              ! r1 <- r0 c v
        opcode = ARITHOFF;
        scode = c;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        off = v              $      &,

move(r1t,r1,r0t,r0,v) =                    ! r1 <- r0 + v
        opcode = ARITHOFF;
        scode = ADD;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        off = v              $      &,

movetag(r1t,r1,r0t,r0,tg,v) =              ! r1 <- tg^(r0v + v)
        opcode = MOVETAG;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        itag = tg;
        off = v              $      &,

push(r1t,r1,r0t,r0,v) =                    ! m(r1) <- r0 + v
```

```
        opcode = PUSH;                  ! r1 <- r1 + 1
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        off = v           $      &,

pushtag(r1t,r1,r0t,r0,tg,v) =           ! m(r1) <- tg^(r0v + v)
        opcode = PUSHTAG;               ! r1 <- r1 + 1
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        itag = tg;
        off = v           $      &,

load(r1t,r1,r0t,r0,v) =                 ! r1 <- m(r0 + v)
        opcode = LOAD;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = v           $      &,

store(r0t,r0,r1t,r1,v) =         ! m(r0 + v) <- r1
        opcode = STORE;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = v           $      &,

br(l) = opcode = BR;                    ! fpc <- l + pc
        longlab = l   $     &,

brind(r1t,r1) =                  ! fpc <- r1
        opcode = BRIND;
        reg1t = r1t;
        reg1 = r1     $     &,

bral(r1t,r1,l) =                 ! fpc <- l + pc
        opcode = BRAL;                  ! r1 <- pc
        reg1t = r1t;
        reg1 = r1;
        lab = l           $      &,

brtageq(tg,r1t,r1,l) =
        opcode = BRTAGEQ;
        reg1t = r1t;
        reg1 = r1;
        itag = tg;
        lab = l           $      &,

brtagneq(tg,r1t,r1,l) =
        opcode = BRTAGNEQ;
        reg1t = r1t;
        reg1 = r1;
        itag = tg;
        lab = l           $      &,
brcond(cond,r1t,r1,l) =
        opcode = BRCOND;
        reg1t = r1t;
        reg1 = r1;
        scode = cond;
        lab = l                 $       &,

brncond(cond,r1t,r1,l) =
        opcode = BRNCOND;
        reg1t = r1t;
        reg1 = r1;
        scode = cond;
        lab = l                 $       &,


brcmp(r1t,r1,r0t,r0,l) =
        opcode = BRCMP;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = l                 $       &,

brncmp(r1t,r1,r0t,r0,l) =
        opcode = BRNCMP;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = l                 $       &,



sbr(l) = opcode = SBR;                          ! fpc <- l + pc
        longlab = l  $      &,

sbrind(r1t,r1) =                        ! fpc <- r1
        opcode = SBRIND;
        reg1t = r1t;
        reg1 = r1    $      &,

sbral(r1t,r1,l) =                       ! fpc <- l + pc
        opcode = SBRAL;                         ! r1 <- pc
        reg1t = r1t;
        reg1 = r1;
        lab = l                 $       &,

sbrtageq(tg,r1t,r1,l) =
        opcode = SBRTAGEQ;
        reg1t = r1t;
        reg1 = r1;
        itag = tg;
        lab = l                 $       &,

sbrtagneq(tg,r1t,r1,l) =
        opcode = SBRTAGNEQ;
        reg1t = r1t;
        reg1 = r1;
        itag = tg;
        lab = l             $       &,

sbrcond(cond,r1t,r1,l) =
        opcode = SBRCOND;
        reg1t = r1t;
        reg1 = r1;
        scode = cond;
        lab = l             $       &,

sbrncond(cond,r1t,r1,l) =
        opcode = SBRNCOND;
        reg1t = r1t;
        reg1 = r1;
        scode = cond;
        lab = l             $       &,

sbrcmp(r1t,r1,r0t,r0,l) =
        opcode = SBRCMP;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = l             $       &,

sbrncmp(r1t,r1,r0t,r0,l) =
        opcode = SBRNCMP;
        reg1t = r1t;
        reg1 = r1;
        reg0t = r0t;
        reg0 = r0;
        imm = l             $       &,


! instructions with lock_unlock same as above except with
! lbit and ulbit appropriately set (not included here for
! sake of brevity)

% now define some macros for instruction sequences %

% getheap(n)       %
getheap(n)  =
        l_arithoff(ADD,G,HP,G,HP,n)
        u_move(T,HP,G,HP,0)  &,

% mem_getheap(n) getheap when HP is in memory %
mem_getheap(n)  =
        load(T,HP,G,HP,0)
        l_nop
        arithoff(ADD,T,B,T,HP,n)
        u_store(G,HP,T,B,0)      &,


makevar(ty,t)        =
        pushtag(T,HP,T,HP,UVAR,0)
        movetag(ty,t,T,HP,BVAR,-1)  &,

puteltcdr(tg,v)      =
        pushtag(T,HP,G,0,tg,v)
        pushtag(T,HP,T,HP,LIST,1)   &,
puteltnil(tg,v)      =
        pushtag(T,HP,G,0,tg,v)
        pushtag(T,HP,G,0,NIL,0)              &,

switchlist(t,niladr,elseadr)       =
        sbrtageq(LIST,T,t,10)
        sbrtageq(NIL,T,t,niladr)
        sbrcond(DRF,T,t,2)
        load(T,t,T,t,0)
        sbr(-5)
        sbrtagneq(UVAR,T,t,elseadr)
        load(T,t,T,t,0)
        nop
        nop
        sbrtageq(UVAR,T,t,elseadr)
        sbr(-11)                  &,

seq_switchlist(t,niladr,elseadr) =
        sbrtageq(LIST,T,t,4)
        sbrtageq(NIL,T,t,niladr)
        sbrtagneq(BVAR,T,t,elseadr)
        load(T,t,T,t,0)
        sbr(-5)                          &,

% deref T reg t %
deref(t)        =
        sbrtageq(UVAR,T,t,3)
        sbrtagneq(BVAR,T,t,8)
        load(T,t,T,t,0)
        sbr(-4)
        brcond(DRF,T,t,2)
        load(T,t,T,t,0)
        sbr(-7)
        nop
        nop
        sbrncond(DRF,T,t,-8)      &,

% seq_deref %
seq_deref(t) =
        sbrtagneq(BVAR,T,t,2)
        load(T,t,T,t,0)
        sbr(-3)                   &,

% wait_ground(t) %
wait_ground(t)      =
        sbrtageq(BVAR,T,t,1)
        sbrtagneq(UVAR,T,t,2)
        load(T,t,T,t,0)
        sbr(-4)                   &      $
```

Example FPPM Code (for `gennrev64`)

```
include ../Soft/fppm.format     $
include ../Soft/fppm.m    $

begin
 .     =     0     $
```

```
initialize:
        move(T,0,G,0,16)
        arithoff(SLL,G,HP,T,0,8)        % initialize HP = 0x1000 %
        move(T,0,G,0,0xf)
        arithoff(SLL,T,E,T,0,12)        % initialize E = 0xf000 %
        move(T,TE,T,E,0)
        nop
        nop
        nop
        nop

main:
        exec(f_main1)
        bral(T,CP,gennrev)
        new

        nop
        nop
        nop
        nop
        nop
        waitexec
        quit

f_main1:
        getheap(8)
        makevar(OUT,2)
        movetag(OUT,1,G,0,INT,64)
        done
        nop

gennrev:
        nop
        exec(s_range)
        br(nrev)
        new

s_range:
        movetag(T,1,G,0,INT,1)
        move(T,TE,T,E,0)
        move(T,2,IN,1,0)
        move(OUT,2,IN,2,0)
        getheap(64)
        movetag(T,7,G,0,INT,63)
        makevar(T,3)
        bral(T,CP,range)
        move(OUT,1,T,3,0)

        done
        nop

range:
        deref(1)
        deref(2)
        arith(SUB,T,0,T,2,T,1)  % T0 <- N - M %
        arithoff(SUB,T,7,T,7,2)
        sbrcond(EQZ,T,0,range1)
        sbrncond(GEZ,T,0,error)
        sbrncond(GEZ,T,7,range_alloc)
```

```
range_noalloc:
        push(T,HP,T,1,0)
        pushtag(T,HP,T,HP,UVAR,0)
        movetag(T,0,T,HP,LIST,-2)
        store(T,3,T,0,0)
        arithoff(ADD,T,1,T,1,1)
        br(range)
        movetag(T,3,T,HP,BVAR,-1)

range_alloc:
        getheap(40)
        br(range_noalloc)
        movetag(T,7,G,0,INT,38)

range1:
        sbrncond(GEZ,T,7,range1_alloc)
range1_noalloc:
        push(T,HP,T,1,0)
        pushtag(T,HP,G,0,NIL,0)
        movetag(T,0,T,HP,LIST,-2)
        brind(T,CP)
        store(T,3,T,0,0)

range1_alloc:
        getheap(8)
        br(range1_noalloc)
        movetag(T,7,G,0,INT,6)

nrev:
        move(T,1,OUT,1,0)
        nop
        switchlist(1,nrev2,error)
        exec(f_nrev11)
        push(T,TE,T,TE,0)             % Save E %
        push(T,TE,T,CP,0)       % save CP %
        push(T,TE,IN,2,0)% save L %
        push(T,TE,OUT,3,0)      % save X %
        push(T,TE,OUT,2,0)      % save L2 %
        bral(T,CP,nrev)
        new

        nop
        exec(sconcat)
        load(OUT,1,T,TE,-1)     % L2 %
        load(OUT,4,T,TE,-2)     % X %
        load(T,CP,T,TE,-4)      % old CP %
        load(OUT,3,T,TE,-3)     % L %
        load(T,TE,T,TE,-5)
        brind(T,CP)
        new

nrev2:
        brind(T,CP)
        store(IN,2,IN,1,0)

f_nrev11:
        move(T,1,IN,1,0)
        getheap(8)
        wait_ground(1)
        load(OUT,1,T,1,1)
        load(OUT,3,T,1,0)
        makevar(OUT,2)
        done
        nop

sconcat:
        move(T,1,OUT,1,0)
        move(T,3,OUT,3,0)
        getheap(40)
        push(T,HP,OUT,4,0)      % [X] %
        pushtag(T,HP,G,0,NIL,0)
        movetag(T,2,T,HP,LIST,-2)
        move(T,7,G,0,38) % count of free locations - 2 %
concat:
        switchlist(1,concat2,error)
        load(T,4,T,1,0)     % X %
        arithoff(SUB,T,7,T,7,2)
        load(T,1,T,1,1)     % L1 %
        sbrncond(GEZ,T,7,concat_alloc)
concat_noalloc:
        push(T,HP,T,4,0)                % X %
        pushtag(T,HP,T,HP,UVAR,0)           % L3 %
        movetag(T,5,T,HP,LIST,-2)
        store(T,3,T,5,0)    % bind the previous location %
        br(concat)
        movetag(T,3,T,HP,BVAR,-1)

concat_alloc:
        getheap(24)
        br(concat_noalloc)
        move(T,7,G,0,22) % count of free locations - 2 %

concat2:
        done
        store(T,3,T,2,0)

error:
        quit

end
```

Appendix 3: Simulator Source Code

A3.1. Overview

The FPPM simulator is written using Zycad's ISP hardware description language and associated simulation tools. Each distinct hardware entity is written in ISP. Instances of these hardware entities and their interconnections are then specified in an *ecology file*. An assembler and a linker create load images of programs which are loaded into simulated into memory before starting the simulations. The decoders of Main Processor and Slave Processor simulators use a microinstruction memory to generate control signals for each pipeline stage. The control words for each instruction are assembled and loaded into simulated microinstruction memory before starting the simulations.

In this appendix, we list the ISP descriptions of the Main Processor (the description of Slave Processor is not listed here; it differs from that of the Main Processor in only a few places), the ideal multi-port shared memory, the update protocol snooping cache and the arbitration unit for the distribution bus and lock. We also list an example ecology file and the control words for the Main Processor. The ISP descriptions and control words use macros defined in a number files that are included by a preprocessing stage. Since the numerical values of these macros are not necessary for an understanding of the simulator, we do not list the macro definitions here. Complete source code listings for the simulator will be made available as a technical report.

A3.2. Main Processor Description

```
/* mainproc.isp
  main processor for FPPM

*/

include(../Inc/global.inc)
include(../Inc/clock.inc)
include(../Inc/mainproc.inc)
include(../Inc/dist_bus.inc)
include(../Inc/task_bus.inc)
include(../Inc/code.inc)
define(anyfail, (fail_latch[0] or fail_latch[1] or fail_latch[2] or
                 fail_latch[3] or fail_latch[4] or fail_latch[5] or
                 fail_latch[6] or fail_latch[7]))
define(exec_done_other, (case $1
                        0: (exec_done7 and exec_done6 and exec_done5 and
                            exec_done4 and exec_done3 and exec_done2 and
                            exec_done1)
                        1: (exec_done7 and exec_done6 and exec_done5 and
                            exec_done4 and exec_done3 and exec_done2 and
                            exec_done0)
                        2: (exec_done7 and exec_done6 and exec_done5 and
                            exec_done4 and exec_done3 and exec_done1 and
                            exec_done0)
                        3: (exec_done7 and exec_done6 and exec_done5 and
                            exec_done4 and exec_done2 and exec_done1 and
                            exec_done0)
                        4: (exec_done7 and exec_done6 and exec_done5 and
                            exec_done3 and exec_done2 and exec_done1 and
                            exec_done0)
                        5: (exec_done7 and exec_done6 and exec_done4 and
                            exec_done3 and exec_done2 and exec_done1 and
                            exec_done0)
                        6: (exec_done7 and exec_done5 and exec_done4 and
                            exec_done3 and exec_done2 and exec_done1 and
                            exec_done0)
                        7: (exec_done6 and exec_done5 and exec_done4 and
                            exec_done3 and exec_done2 and exec_done1 and
                            exec_done0)
                       esac))
define(exec_done_set, (case $1
                        0: exec_done0
                        1: exec_done1
                        2: exec_done2
                        3: exec_done3
                        4: exec_done4
                        5: exec_done5
                        6: exec_done6
                        7: exec_done7
                       esac))

define(other_fail, (case $1
                0: fail_latch[1] or fail_latch[2] or fail_latch[3] or
                   fail_latch[4] or fail_latch[5] or fail_latch[6] or
                   fail_latch[7]
                1: fail_latch[0] or fail_latch[2] or fail_latch[3] or
                   fail_latch[4] or fail_latch[5] or fail_latch[6] or
```

```
                    fail_latch[7]
                2: fail_latch[0] or fail_latch[1] or fail_latch[3] or
                    fail_latch[4] or fail_latch[5] or fail_latch[6] or
                    fail_latch[7]
                3: fail_latch[0] or fail_latch[1] or fail_latch[2] or
                    fail_latch[4] or fail_latch[5] or fail_latch[6] or
                    fail_latch[7]
                4: fail_latch[0] or fail_latch[1] or fail_latch[2] or
                    fail_latch[3] or fail_latch[5] or fail_latch[6] or
                    fail_latch[7]
                5: fail_latch[0] or fail_latch[1] or fail_latch[2] or
                    fail_latch[3] or fail_latch[4] or fail_latch[6] or
                    fail_latch[7]
                6: fail_latch[0] or fail_latch[1] or fail_latch[2] or
                    fail_latch[3] or fail_latch[4] or fail_latch[5] or
                    fail_latch[7]
                7: fail_latch[0] or fail_latch[1] or fail_latch[2] or
                    fail_latch[3] or fail_latch[4] or fail_latch[5] or
                    fail_latch[6]

                esac))

port
    clock           'input,

    i_maddr<ADDR> 'output:connect,
    i_mdata<WORD> 'bidirectional:disconnect,
    i_mrq (0)     'output:connect,
    i_mrw (READ)      'output:connect,
    i_mrdy            'input,

    maddr<ADDR>       'output:connect,
    mdata<WORD>       'bidirectional:disconnect,
    mrq (0)           'output:connect,
    mrw (READ)        'output:connect,
    mpush (0)   'output:connect,
    mrdv        'input,

    free_exec1 (1)    'bidirectional:and:connect,
    free_exec2 (1)    'bidirectional:and:connect,
    free_exec3 (1)    'bidirectional:and:connect,
    free_exec4 (1)    'bidirectional:and:connect,
    free_exec5 (1)    'bidirectional:and:connect,
    free_exec6 (1)    'bidirectional:and:connect,
    free_exec7 (1)    'bidirectional:and:connect,

    task_bus<TB_WID>(0) 'output:connect,

    dist_bus<DB_WID> 'bidirectional:or:connect,
    db_rq (0)   'output:connect,
    db_gr       'input,

    lock_rq (0)  'output:connect,
    lock_rel (1) 'output:connect,
    lock_gr           'input,

    fail_sig0 (0)'bidirectional:or:connect,
    fail_sig1 (0)'bidirectional:or:connect,
    fail_sig2 (0)'bidirectional:or:connect,
        fail_sig3 (0)'bidirectional:or:connect,
        fail_sig4 (0)'bidirectional:or:connect,
        fail_sig5 (0)'bidirectional:or:connect,
        fail_sig6 (0)'bidirectional:or:connect,
        fail_sig7 (0)'bidirectional:or:connect,

        clear_sig0 (0)      'bidirectional:or:connect,
        clear_sig1 (0)      'bidirectional:or:connect,
        clear_sig2 (0)      'bidirectional:or:connect,
        clear_sig3 (0)      'bidirectional:or:connect,
        clear_sig4 (0)      'bidirectional:or:connect,
        clear_sig5 (0)      'bidirectional:or:connect,
        clear_sig6 (0)      'bidirectional:or:connect,
        clear_sig7 (0)      'bidirectional:or:connect,

        exec_done0 (1)      'bidirectional:and:connect,
        exec_done1 (1)      'bidirectional:and:connect,
        exec_done2 (1)      'bidirectional:and:connect,
        exec_done3 (1)      'bidirectional:and:connect,
        exec_done4 (1)      'bidirectional:and:connect,
        exec_done5 (1)      'bidirectional:and:connect,
        exec_done6 (1)      'bidirectional:and:connect,
        exec_done7 (1)      'bidirectional:and:connect,

        pref_set<2:0> (0)'bidirectional:connect,

        pref_stall    'input:or;

state
        d_inst_reg<WORD>, d_inst_reg_0<WORD>,
        a_inst_reg<WORD>, a_inst_reg_0<WORD>,
        m_inst_reg<WORD>, m_inst_reg_0<WORD>,
        w_inst_reg<WORD>, w_inst_reg_0<WORD>,

        a_uinst<UINST_WID>, a_uinst_0<UINST_WID>,
        m_uinst<UINST_WID>, m_uinst_0<UINST_WID>,
        w_uinst<UINST_WID>, w_uinst_0<UINST_WID>,

        f_pc<ADDR>, f_pc_0<ADDR>,
        d_pc<ADDR>, d_pc_0<ADDR>,
        a_pc<ADDR>, a_pc_0<ADDR>,
        m_pc<ADDR>, m_pc_0<ADDR>,
        w_pc<ADDR>, w_pc_0<ADDR>,

        f_current_set<3:0>, f_current_set_0<3:0>,
        d_current_set<3:0>, d_current_set_0<3:0>,
        a_current_set<3:0>, a_current_set_0<3:0>,
        m_current_set<3:0>, m_current_set_0<3:0>,
        w_current_set<3:0>, w_current_set_0<3:0>,

        f_stage_valid, f_stage_valid_0,
        d_stage_valid, d_stage_valid_0,
        a_stage_valid, a_stage_valid_0,
        m_stage_valid, m_stage_valid_0,
        w_stage_valid, w_stage_valid_0,

        m_done, m_done_0,
        w_done, w_done_0,
        treg[REG_RANGE]<WORD>,        /* temp regs */
        greg[REG_RANGE]<WORD>,        /* global regs */

        reg[SET_RANGE][REG_RANGE]<WORD>,    /* arg regs */
        reg_v[SET_RANGE][REG_RANGE],

        temp_reg0<WORD>,
        temp_reg1<WORD>,
        temp_reg2<WORD>,

        reg0<WORD>, reg0_0<WORD>,
        reg1<WORD>, reg1_0<WORD>,
        reg2<WORD>, reg2_0<WORD>,

        temp_reg0_v,
        temp_reg1_v,
        temp_reg2_v,

        lock_granted (0),
        lock_set<2:0>,

        condition,
        branch_address<ADDR>,
        decoder_output<UINST_WID>,

        mar<WORD>, mar_0<WORD>,
        mdr<WORD>, mdr_0<WORD>,
        mdr_in<WORD>, mdr_in_0<WORD>,
        mdr_in_temp<WORD>, mdr_in_temp_0<WORD>,
        result<WORD>, result_0<WORD>,

        b_operand<WORD>,

        alu_output<WORD>,
        inc_output<WORD>,

        stall_pipe,

        exec_num<2:0>,
        last_exec[0:1]<2:0>,
        max_exec<3:0>,

        dist_bus_latch<DB_WID>,
        fail_enable (1), fail_enable_0,
        fail_latch[SET_RANGE],

        squash,

        inst_count<WORD> (0),        /* number of instructions executed */

        stall_f, stall_d, stall_m, stall_w;


memory
        uinst_mem[0:0xff]<UINST_WID>;

format
        d_fpcctl        =       decoder_output<1:0>,
        d_branchlength  =       decoder_output<3:2>,
        d_cond          =       decoder_output<6:4>,
        d_dispatch  =     decoder_output<8:7>,
        d_squash    =     decoder_output<10:9>,
        d_regset    =     decoder_output<12:11>,
        d_fail      =     decoder_output<14:13>,
        d_stall         =       decoder_output<18:15>,

        a_alucode   =     a_uinst<19:19>,
        a_bopsrc    =     a_uinst<20:20>,
        a_tagsrc    =     a_uinst<21:21>,
        a_bypass    =     a_uinst<22:22>,
        a_incsrc    =     a_uinst<23:23>,
        a_result    =     a_uinst<24:24>,
        a_marsrc    =     a_uinst<25:25>,
        a_mdrsrc    =     a_uinst<26:26>,

        m_ctrl          =       m_uinst<28:27>,
        m_bypass    =     m_uinst<29:29>,

        w_writereg  =     w_uinst<31:30>;

format      ! for decoder bypasses
        m_abypass =       m_uinst<22:22>,
        w_mbypass =       w_uinst<29:29>;

format
        a_alucode_0     =       a_uinst_0<19:19>,
        a_bopsrc_0 =      a_uinst_0<20:20>,
        a_tagsrc_0  =     a_uinst_0<21:21>,
        a_bypass_0 =      a_uinst_0<22:22>,
        a_incsrc_0  =     a_uinst_0<23:23>,
        a_result_0  =     a_uinst_0<24:24>,
        a_marsrc_0 =      a_uinst_0<25:25>,
        a_mdrsrc_0 =      a_uinst_0<26:26>,

        m_ctrl_0    =     m_uinst_0<28:27>,
        m_bypass_0      =       m_uinst_0<29:29>,

        w_writereg_0    =       w_uinst_0<31:30>;

/******** useful function definitions ********************************/

function exec_done_previous(set<2:0>)<0:0> :=
(
   state i<2:0>, done;

   i = set - 1;
   done = 1;
   next;

   while (i neq pref_set)
   (
       done = done and exec_done_set(i);
       i = i + 1;
   );
   exec_done_previous = done;
)

/************************ Fetch Stage ********************************/
```

```
when f_stage_slave (clock:PHI_0):=
(
  f_pc_0 = f_pc;
  f_current_set_0 = f_current_set;
)

when drive_i_addr (clock:PHI_0):=
(
      i_maddr = f_pc;
      i_mrq = 1;
      delay(0);
)

when read_i_data(clock:PHI_1):=
(
      if i_mrdy
      (
        i_mrq = 0;
        delay(ND);
        if ((anyfail) and (fail_enable_0)) or (squash and not stall_pipe)
        (
            d_inst_reg = 0;
            d_stage_valid = 0;
        )
        else if not stall_pipe
        (
            d_inst_reg = i_mdata;
            d_pc = f_pc_0 + 1;
            d_current_set = f_current_set_0;
            d_stage_valid = 1;
        )
      )
)

/************************ Decode Stage ************************************/
when d_stage_slave(clock:PHI_0):=
(
  d_pc_0 = d_pc;
  d_inst_reg_0 = d_inst_reg;
  d_current_set_0 = d_current_set;
  d_stage_valid_0 = d_stage_valid;
)

when manage_fpc(clock:PHI_1):=
(
      delay(ND);
      if (anyfail) and (fail_enable_0)
      (
        f_pc = greg[FAIL_REG]<VALUE>;
        fail_enable = 0;
      )
      else if not stall_pipe
      (
        case d_fpcctl
        DF_INC: f_pc = f_pc_0 + 1
        DF_BRANCH: f_pc = branch_address
        DF_COND: if (condition)
                       f_pc = branch_address
                 else f_pc = f_pc_0 + 1
            DF_REG1: f_pc = temp_reg1<VALUE>
            esac;
        );
)

when manage_fcurrentset(clock:PHI_1):=
(
        state temp_clear[SET_RANGE];

        temp_clear[0] = 0;
        temp_clear[1] = 0;
        temp_clear[2] = 0;
        temp_clear[3] = 0;
        temp_clear[4] = 0;
        temp_clear[5] = 0;
        temp_clear[6] = 0;
        temp_clear[7] = 0;

        delay(ND);
        if (not stall_pipe) and (not ((anyfail) and fail_enable_0))
        (
          case d_regset
          DR_NEW:
          (
              f_current_set = (f_current_set_0 + 1) mod NUM_SET;
              temp_clear[((f_current_set_0 ext SET_MAX) + 1) mod NUM_SET]= 1;
          )
          DR_OLD:
          (
              f_current_set = ((f_current_set_0 ext SET_MAX) + NUM_SET - 1)
                      mod NUM_SET;
          )
          esac;
        );
        next;

        pref_set = f_current_set;
        clear_sig0 = temp_clear[0];
        clear_sig1 = temp_clear[1];
        clear_sig2 = temp_clear[2];
        clear_sig3 = temp_clear[3];
        clear_sig4 = temp_clear[4];
        clear_sig5 = temp_clear[5];
        clear_sig6 = temp_clear[6];
        clear_sig7 = temp_clear[7];
)

when lock_request(clock:PHI_0):=
(
 lock_rq = d_inst_reg<LBIT>;

  if lock_gr
  (
      lock_granted = 1;
      lock_set = d_current_set_0;
v   )
)

when clear_cp (clock:PHI_1):=
(
  if d_fail eql DL_CLEARCP
  (
     delay(ND);

     if not stall_pipe
       reg_v[d_current_set_0][L] = 0;
  )
)

when decode_bypass (clock:PHI_0):=
(
  case d_inst_reg<REG0T>
  IN,OUT:
  (
     temp_reg0 = reg[(d_current_set +
     (d_inst_reg<REG0T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG0>];
     temp_reg0_v = reg_v[(d_current_set +
     (d_inst_reg<REG0T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG0>];
  )
  G:
  (
     temp_reg0 = greg[d_inst_reg<REG0>];
     temp_reg0_v = 1;
  )
  T:
  (
     if m_abypass and (m_inst_reg<REG1> eql d_inst_reg<REG0>)
       temp_reg0 = result
     else if w_mbypass and (w_inst_reg<REG1> eql d_inst_reg<REG0>)
       temp_reg0 = mdr_in
     else
       temp_reg0 = treg[d_inst_reg<REG0>];
     temp_reg0_v = 1;
  )
  esac;

  case d_inst_reg<REG1T>
  IN,OUT:
  (
     temp_reg1 = reg[(d_current_set +
     (d_inst_reg<REG1T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG1>];
     temp_reg1_v = reg_v[(d_current_set +
     (d_inst_reg<REG1T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG1>];
  )
  G:
  (
     temp_reg1 = greg[d_inst_reg<REG1>];
     temp_reg1_v = 1;
  )
  T:
  (
     if m_abypass and (m_inst_reg<REG1> eql d_inst_reg<REG1>)
       temp_reg1 = result
     else if w_mbypass and (w_inst_reg<REG1> eql d_inst_reg<REG1>)
       temp_reg1 = mdr_in
     else
       temp_reg1 = treg[d_inst_reg<REG1>];
     temp_reg1_v = 1;
```

```
        )
        esac;

        case d_inst_reg<REG2T>
        IN,OUT:
        (
            temp_reg2 = reg[(d_current_set +
            (d_inst_reg<REG2T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG2>];
            temp_reg2_v = reg_v[(d_current_set +
            (d_inst_reg<REG2T> ext SET_MAX)) mod NUM_SET][d_inst_reg<REG2>];
        )
        G:
        (
            temp_reg2 = greg[d_inst_reg<REG2>];
            temp_reg2_v = 1;
        )
        T:
        (
            if m_abypass and (m_inst_reg<REG1> eql d_inst_reg<REG2>)
               temp_reg2 = result
            else if w_mbypass and (w_inst_reg<REG1> eql d_inst_reg<REG2>)
               temp_reg2 = mdr_in
            else
               temp_reg2 = treg[d_inst_reg<REG2>];
            temp_reg2_v = 1;
        )
        esac;

        next;

        wait(clock:PHI_1);

        delay(ND);
        if not stall_pipe
        (
            reg0 = temp_reg0;
            reg1 = temp_reg1;
            reg2 = temp_reg2;
        );
    )

    when decoder (clock:PHI_0):=
    (
        case d_inst_reg<OPCODE>
        NOP: decoder_output = uinst_mem[uNOP]
        QUIT: decoder_output = uinst_mem[uQUIT]
        NEW: decoder_output = uinst_mem[uNEW]
        FAIL: decoder_output = uinst_mem[uFAIL]
        DONE: decoder_output = uinst_mem[uERROR]
        EXEC: decoder_output = uinst_mem[uEXEC]
        WAITEXEC: decoder_output = uinst_mem[uWAITEXEC]
        CLEARFAIL: decoder_output = uinst_mem[uCLEARFAIL]
        CLEARCP: decoder_output = uinst_mem[uCLEARCP]
        OLD: decoder_output = uinst_mem[uOLD]

        ARITH:     case d_inst_reg<REG1T>
               T: decoder_output = uinst_mem[uARITH]
               default: decoder_output = uinst_mem[uARITHG]
               esac
    ARITHOFF:case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uARITHOFF]
        default: decoder_output = uinst_mem[uARITHOFFG]
        esac
    MOVETAG:case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uMOVETAG]
        default: decoder_output = uinst_mem[uMOVETAGG]
        esac
    PUSH:      case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uPUSH]
        default: decoder_output = uinst_mem[uPUSHG]
        esac
    PUSHTAG:       case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uPUSHTAG]
        default: decoder_output = uinst_mem[uPUSHTAGG]
        esac
    LOAD:      case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uLOAD]
        default: decoder_output = uinst_mem[uLOADG]
        esac
    STORE: decoder_output = uinst_mem[uSTORE]

    BR: decoder_output = uinst_mem[uBR]
    BRIND: decoder_output = uinst_mem[uBRIND]
    BRAL:      case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uBRAL]
        default: decoder_output = uinst_mem[uBRALG]
        esac
    BRTAGEQ: decoder_output = uinst_mem[uBRTAGEQ]
    BRTAGNEQ: decoder_output = uinst_mem[uBRTAGNEQ]
    BRCOND: decoder_output = uinst_mem[uBRCOND]
    BRNCOND: decoder_output = uinst_mem[uBRNCOND]
    BRCMP: decoder_output = uinst_mem[uBRCMP]
    BRNCMP: decoder_output = uinst_mem[uBRNCMP]


    SBR: decoder_output = uinst_mem[uSBR]
    SBRIND: decoder_output = uinst_mem[uSBRIND]
    SBRAL:     case d_inst_reg<REG1T>
        T: decoder_output = uinst_mem[uSBRAL]
        default: decoder_output = uinst_mem[uSBRALG]
        esac
    SBRTAGEQ: decoder_output = uinst_mem[uSBRTAGEQ]
    SBRTAGNEQ: decoder_output = uinst_mem[uSBRTAGNEQ]
    SBRCOND: decoder_output = uinst_mem[uSBRCOND]
    SBRNCOND: decoder_output = uinst_mem[uSBRNCOND]
    SBRCMP: decoder_output = uinst_mem[uSBRCMP]
    SBRNCMP: decoder_output = uinst_mem[uSBRNCMP]

    default: decoder_output = uinst_mem[uERROR]
    esac;
)

when d_branch_compute(clock:PHI_0):=
(
  delay(ND); ! wait for decoder output
  case d_branchlength
  DB_LAB: branch_address = d_pc + d_inst_reg<LAB>
  DB_LONGLAB: branch_address = d_pc + d_inst_reg<LONGLAB>
  DB_IMM: branch_address = d_pc + d_inst_reg<IMM>
  esac
)

when d_compute_condition(clock:PHI_0):=
(
  delay(ND);      ! wait until registers are read
  case d_cond
  DC_COND: condition =
      (case d_inst_reg<SCODE>
      GEZ: temp_reg1<VALUE> geq 0
      EQZ: temp_reg1<VALUE> eql 0
      DRF: not((temp_reg1<TAG> eql BVAR) or
              ((temp_reg1<TAG> eql UVAR) and
               not (exec_done_previous(d_current_set))))
      OTF: other_fail(d_current_set)
      CPV: reg_v[0][L] or reg_v[1][L] or reg_v[2][L] or reg_v[3][L] or
              reg_v[4][L] or reg_v[5][L] or reg_v[6][L] or reg_v[7][L]
      VCP: reg_v[d_current_set][L]
      esac)
  DC_NCOND: condition =
      not (case d_inst_reg<SCODE>
      GEZ: temp_reg1<VALUE> geq 0
      EQZ: temp_reg1<VALUE> eql 0
      DRF: not((temp_reg1<TAG> eql BVAR) or
              ((temp_reg1<TAG> eql UVAR) and
               (exec_done_previous(d_current_set))))
      OTF: other_fail(d_current_set)
      CPV: reg_v[0][L] or reg_v[1][L] or reg_v[2][L] or reg_v[3][L] or
              reg_v[4][L] or reg_v[5][L] or reg_v[6][L] or reg_v[7][L]
      VCP: reg_v[d_current_set][L]
      esac)
  DC_TAGEQ: condition = (d_inst_reg<ITAG> eql temp_reg1<TAG>)
  DC_TAGNEQ: condition = (d_inst_reg<ITAG> neq temp_reg1<TAG>)
  DC_CMP: condition = (temp_reg1 eql temp_reg0)
  DC_NCMP: condition = (temp_reg1 neq temp_reg0)
  esac
)

when d_microsequencer(clock:PHI_1):=
(
  delay(ND);
  if ((anyfail) and fail_enable_0)
  (
      a_uinst = 0;
      a_stage_valid = 0;
  )
  else if not stall_pipe
  (
      a_uinst = decoder_output;
      a_inst_reg = d_inst_reg_0;
      a_pc = d_pc_0;
      a_current_set = d_current_set_0;
      a_stage_valid = d_stage_valid_0;
      inst_count = inst_count + (d_stage_valid_0 ext 32);
  );
)

when pull_free_execs(clock:PHI_0) :=
(
   if (last_exec[0] eql 1) or (last_exec[1] eql 1) free_exec1 = 0
       else (if (max_exec geq 1) free_exec1 = 1);
   if (last_exec[0] eql 2) or (last_exec[1] eql 2) free_exec2 = 0
       else (if (max_exec geq 2) free_exec2 = 1);
   if (last_exec[0] eql 3) or (last_exec[1] eql 3) free_exec3 = 0
       else (if (max_exec geq 3) free_exec3 = 1);
   if (last_exec[0] eql 4) or (last_exec[1] eql 4) free_exec4 = 0
       else (if (max_exec geq 4) free_exec4 = 1);
   if (last_exec[0] eql 5) or (last_exec[1] eql 5) free_exec5 = 0
       else (if (max_exec geq 5) free_exec5 = 1);
   if (last_exec[0] eql 6) or (last_exec[1] eql 6) free_exec6 = 0
       else (if (max_exec geq 6) free_exec6 = 1);
   if (last_exec[0] eql 7) or (last_exec[1] eql 7) free_exec7 = 0
       else (if (max_exec geq 7) free_exec7 = 1);
)

when set_exec_num(clock:PHI_1):=
(
   if free_exec1
       exec_num = 1
   else if free_exec2
       exec_num = 2
   else if free_exec3
       exec_num = 3
   else if free_exec4
       exec_num = 4
   else if free_exec5
       exec_num = 5
   else if free_exec6
       exec_num = 6
   else if free_exec7
       exec_num = 7
   else
       exec_num = 0;
   next;
)

when d_dispatch_task(clock:PHI_1):=
(
   delay(ND);

   if not (stall_pipe or ((anyfail)and fail_enable_0))
   (
       case d_dispatch
       DD_NONE:
       (
            task_bus = 0;
            last_exec[1] = last_exec[0];
            last_exec[0] = 0;
       )
       DD_EXEC:
       (
          last_exec[1] = last_exec[0];
          last_exec[0] = exec_num;
          task_bus<TB_UNIT> = exec_num;
          task_bus<TB_ADDRESS> = branch_address;
          task_bus<TB_SET> = d_current_set_0;
       )
```

```
        esac
    )
    else
    (
        task_bus = 0;
        last_exec[1] = last_exec[0];
        last_exec[0] = 0;
    )
)

when d_squash_gen(clock:PHI_0):=
(
    delay(2*ND); ! wait for condition to be computed

    case d_squash
    DQ_NO: squash = 0
    DQ_YES: squash = 1
    DQ_COND: squash = condition
    esac
)

when set_fail_sigs(clock:PHI_1):=
(
        state temp_fail[0:7];

        temp_fail[0] = 0;
        temp_fail[1] = 0;
        temp_fail[2] = 0;
        temp_fail[3] = 0;
        temp_fail[4] = 0;
        temp_fail[5] = 0;
        temp_fail[6] = 0;
        temp_fail[7] = 0;

        delay(ND);
        if (d_fail eql DL_FAIL) and (not stall_pipe)
        (
           temp_fail[d_current_set_0] = 1;
        );
        next;

        fail_sig0 = temp_fail[0];
        fail_sig1 = temp_fail[1];
        fail_sig2 = temp_fail[2];
        fail_sig3 = temp_fail[3];
        fail_sig4 = temp_fail[4];
        fail_sig5 = temp_fail[5];
        fail_sig6 = temp_fail[6];
        fail_sig7 = temp_fail[7];
)

when latch_fail (clock:PHI_0):=
(
        fail_enable_0 = fail_enable;
        if fail_sig0 fail_latch[0] = 1;
        if fail_sig1 fail_latch[1] = 1;
        if fail_sig2 fail_latch[2] = 1;
        if fail_sig3 fail_latch[3] = 1;
        if fail_sig4 fail_latch[4] = 1;
```

```
          if fail_sig5 fail_latch[5] = 1;
          if fail_sig6 fail_latch[6] = 1;
          if fail_sig7 fail_latch[7] = 1;

          wait(clock:PHI_1);

          delay(ND);
          if (d_fail eql DL_CLEAR) and (not stall_pipe)
          (
               fail_latch[d_current_set] = 0;
               fail_enable = 1;
          );
     )

     when set_done_signals(clock:PHI_0):=
     (
       state temp_set[SET_RANGE];

       temp_set[0] = 1;
       temp_set[1] = 1;
       temp_set[2] = 1;
       temp_set[3] = 1;
       temp_set[4] = 1;
       temp_set[5] = 1;
       temp_set[6] = 1;
       temp_set[7] = 1;

       next;

       temp_set[f_current_set] = 0;
       if d_stage_valid
            temp_set[d_current_set] = 0;
       if m_stage_valid
            temp_set[m_current_set] = 0;
       if w_stage_valid
            temp_set[w_current_set] = 0;

       wait(clock:PHI_1);

       exec_done0 = temp_set[0];
       exec_done1 = temp_set[1];
       exec_done2 = temp_set[2];
       exec_done3 = temp_set[3];
       exec_done4 = temp_set[4];
       exec_done5 = temp_set[5];
       exec_done6 = temp_set[6];
       exec_done7 = temp_set[7];
     )

     /********************* Address Computation Stage *************************/
     when a_slave (clock:PHI_0):=
     (
          a_inst_reg_0 = a_inst_reg;
          a_uinst_0 = a_uinst;
          a_pc_0 = a_pc;
          a_current_set_0 = a_current_set;
          a_stage_valid_0 = a_stage_valid;
     )
```

```
when address_bypass (clock:PHI_0):=
(
  case a_inst_reg<REG0T>
  IN,OUT,G:
  (
     reg0_0 = reg0;

  )
  T:
  (
     if m_abypass and (m_inst_reg<REG1> eql a_inst_reg<REG0>)
       reg0_0 = result
     else if w_mbypass and (w_inst_reg<REG1> eql a_inst_reg<REG0>)
       reg0_0 = mdr_in
     else
       reg0_0 = reg0;
  )
  esac;

  case a_inst_reg<REG1T>
  IN,OUT,G:
  (
     reg1_0 = reg1;

  )
  T:
  (
     if m_abypass and (m_inst_reg<REG1> eql a_inst_reg<REG1>)
       reg1_0 = result
     else if w_mbypass and (w_inst_reg<REG1> eql a_inst_reg<REG1>)
       reg1_0 = mdr_in
     else
       reg1_0 = reg1;
  )
  esac;

  case a_inst_reg<REG2T>
  IN,OUT,G:
  (
     reg2_0 = reg2;

  )
  T:
  (
     if m_abypass and (m_inst_reg<REG1> eql a_inst_reg<REG2>)
       reg2_0 = result
     else if w_mbypass and (w_inst_reg<REG1> eql a_inst_reg<REG2>)
       reg2_0 = mdr_in
     else
       reg2_0 = reg2;
  )
  esac;
)


when a_microsequencer (clock:PHI_1):=
(
  delay(ND);
  if ((anyfail) and fail_enable_0)
  (
      m_uinst = 0;
      m_stage_valid = 0;
  )
  else if not stall_pipe
  (
      m_uinst = a_uinst_0;
      m_inst_reg = a_inst_reg_0;
      m_pc = a_pc_0;
      m_current_set = a_current_set_0;
      m_stage_valid = a_stage_valid_0;
      m_done = 0;
  )
)

when incrementer(clock:PHI_0):=
(
  delay(ND);

  case a_incsrc
  AI_R1:
  (
      inc_output<VALUE> = reg1_0<VALUE> + 1;
      inc_output<TAG> = reg1_0<TAG>;
  )
  AI_PC:
  (
      inc_output<VALUE> = a_pc + 1;
  )
  esac;
)

when alu(clock:PHI_0):=
(
  delay(ND);

  case a_bopsrc
  AB_OFF:
  (
      b_operand<VALUE> = a_inst_reg<OFF> sxt 28;
      b_operand<TAG> = a_inst_reg<ITAG>;
  )
  AB_R2: b_operand = reg2_0
  esac;
  next;

  case a_alucode
  AA_ADD: alu_output<VALUE> = reg0_0<VALUE> + b_operand<VALUE>
  AA_SCODE:
  (
      alu_output<VALUE> =
      case a_inst_reg<SCODE>
      ADD: reg0_0<VALUE> + b_operand<VALUE>
      SUB: reg0_0<VALUE> - b_operand<VALUE>
      NAND: reg0_0<VALUE> nand b_operand<VALUE>
      NOR: reg0_0<VALUE> nor b_operand<VALUE>
      SLL: reg0_0<VALUE> *: logical b_operand<VALUE>
      SRL: reg0_0<VALUE> /: logical b_operand<VALUE>
      MAX: case (reg0_0<VALUE> geq b_operand<VALUE>)
```

```
                0: b_operand<VALUE>
                1: reg0_0<VALUE>
                esac
            esac
        )
        esac;

        case a_tagsrc
        AT_R0: alu_output<TAG> = reg0_0<TAG>
        AT_ITAG: alu_output<TAG> = a_inst_reg<ITAG>
        esac;
    )

    when source_mar(clock:PHI_1):=
    (
        delay(ND);

        if not stall_pipe
        (
            case a_marsrc_0
            AM_ALU: mar = alu_output
            AM_R1: mar = reg1_0
            esac;
        )
    )

    when source_mdr(clock:PHI_1):=
    (
        delay(ND);

        if not stall_pipe
        (
            case a_mdrsrc_0
            AD_ALU: mdr = alu_output
            AD_R1: mdr = reg1_0
            esac;
        )
    )


    when source_result(clock:PHI_1):=
    (
        delay(ND);

        if not stall_pipe
        (
            case a_result_0
            AR_ALU: result = alu_output
            AR_INC: result = inc_output
            esac;
        )
    )

    /************************ Memory stage ******************************/
    when m_copy (clock:PHI_0):=
    (
            m_inst_reg_0 = m_inst_reg;
            m_uinst_0 = m_uinst;
            m_pc_0 = m_pc;
```

```
          m_current_set_0 = m_current_set;
          m_stage_valid_0 = m_stage_valid;
          mar_0 = mar;
          mdr_0 = mdr;
          result_0 = result;
          m_done_0 = m_done;
          mdr_in_temp_0 = mdr_in_temp;
)

when m_microsequencer (clock:PHI_1):=
(
   delay(ND);
   if ((anyfail) and fail_enable_0)
   (
        w_uinst = 0;
        w_stage_valid = 0;
   )
   else if not stall_pipe
   (
        w_uinst = m_uinst_0;
        w_inst_reg = m_inst_reg_0;
        w_pc = m_pc_0;
        w_current_set = m_current_set_0;
        w_stage_valid = m_stage_valid_0;
        w_done = 0;
   );
)

when drive_mem_addr(clock:PHI_0):=
(
        maddr = mar<ADDR>;

   if not m_done
   (

        case m_ctrl
        MC_READ:
        (
           mrw = READ;
           mrq = 1;
        )
        MC_WRITE:
        (
           vconnect(mdata,mdr);
           mrw = WRITE;
           mpush = 0;
           mrq = 1;
        )
        MC_PUSH:
        (
           vconnect(mdata,mdr);
           mrw = WRITE:
           mpush = 1;
           mrq = 1;
        )
        default:
        (
           mrw = READ;
           mrq = 0;
```

```
            disconnect(mdata);
         )
         esac;
   )
   else
   (
         mrw = READ;
         mrq = 0;
         disconnect(mdata);
   );
   delay(0);
)

when handle_mdata(clock:PHI_1):=
(
   delay(ND);
   case m_ctrl_0
   MC_READ:
   (
         if m_done_0
         (
            if not stall_pipe
            (
                  mdr_in = mdr_in_temp_0;
            )
         )
         else if mrdy
         (
            mrq = 0;
            if not stall_pipe
            (
                  mdr_in = mdata;
            )
            else
            (
                  mdr_in_temp = mdata;
                  m_done = 1;
            )
         )
   )
   MC_PUSH,
   MC_WRITE:
   (
         if mrdy
         (
            mrq = 0;
            disconnect(mdata);
            if stall_pipe
                  m_done = 1;
         );
         if not stall_pipe
            mdr_in = result_0;
   )
   default:
   (
         mrq = 0;
         disconnect(mdata);
         if not stall_pipe
            mdr_in = result_0;
```

```
  )
  esac;
)

/********************* Writeback stage **********************************/
when w_slave_copy (clock:PHI_0):=
(
     w_uinst_0 = w_uinst;
     w_inst_reg_0 = w_inst_reg;
     mdr_in_0 = mdr_in;
     w_current_set_0 = w_current_set;
     w_stage_valid = w_stage_valid_0;
     w_pc_0 = w_pc;
     w_done_0 = w_done;
)

when write_register(clock:PHI_0):=
(
  case w_writereg
  WW_NO:
  (
     db_rq = 0;
     delay(0);

     wait(clock:PHI_1);

     disconnect(dist_bus);
     delay(0);
  )
  WW_LOCAL:
  (
     db_rq = 0;
     delay(0);

     wait(clock:PHI_1);

     disconnect(dist_bus);
     delay(ND);
     if not stall_pipe
     (
        treg[w_inst_reg_0<REG1>] = mdr_in_0;
     )
  )
  WW_GLOBAL:
  (
     db_rq = 1 and not w_done;
     delay(0);

     wait(clock:PHI_1);

     if db_gr and (not fail_latch[w_current_set_0])
     (
        connect(dist_bus);
        delay(0);
        dist_bus<DB_CODE> = WRITE;
        dist_bus<DB_TYPE> = w_inst_reg_0<REG1T>;
        dist_bus<DB_DATA> = mdr_in_0;
        dist_bus<DB_REG> = w_inst_reg_0<REG1>;
        dist_bus<DB_SET> = (w_current_set_0 +
                    (w_inst_reg_0<REG1T> ext SET_MAX)) mod NUM_SET;
            delay(ND);
            if stall_pipe
                w_done = 1;
        )
        else
        (
            disconnect(dist_bus);
            delay(0);
        )
    )
    esac;
)

when write_global_registers(clock:PHI_0):=
(
    dist_bus_latch = dist_bus;

    wait(clock:PHI_1);

    if (dist_bus_latch<DB_CODE> eql WRITE) and
        (not case dist_bus_latch<DB_SET>
                0: fail_latch[7]
                1: fail_latch[0]
                2: fail_latch[1]
                3: fail_latch[2]
                4: fail_latch[3]
                5: fail_latch[4]
                6: fail_latch[5]
                7: fail_latch[6]
          esac)
    (
        case dist_bus_latch<DB_TYPE>
        IN,OUT:
        (
            reg[dist_bus_latch<DB_SET>][dist_bus_latch<DB_REG>] =
                dist_bus_latch<DB_DATA>;
            reg_v[dist_bus_latch<DB_SET>][dist_bus_latch<DB_REG>] = 1;
        )
        G:
        (
            greg[dist_bus_latch<DB_REG>] = dist_bus_latch<DB_DATA>;
        )
        esac;
    );
)

when clear_registersets(clock:PHI_0):=
(
    state temp_clear[SET_RANGE], i<3:0>;

    temp_clear[0] = clear_sig7 or fail_sig7;
    temp_clear[1] = clear_sig0 or fail_sig0;
    temp_clear[2] = clear_sig1 or fail_sig1;
    temp_clear[3] = clear_sig2 or fail_sig2;
    temp_clear[4] = clear_sig3 or fail_sig3;
    temp_clear[5] = clear_sig4 or fail_sig4;
    temp_clear[6] = clear_sig5 or fail_sig5;
    temp_clear[7] = clear_sig6 or fail_sig6;
```

```
wait(clock:PHI_1);

if temp_clear[0]
(
    i = 0; next;
    do
    (
      reg_v[0][i] = 0;
      i = i + 1;
    ) until (i eql 0);
);

if temp_clear[1]
(
    i = 0; next;
    do
    (
      reg_v[1][i] = 0;
      i = i + 1;
    ) until (i eql 0);
);

if temp_clear[2]
(
    i = 0; next;
    do
    (
      reg_v[2][i] = 0;
      i = i + 1;
    ) until (i eql 0);
);

if temp_clear[3]
(
    i = 0; next;
    do
    (
      reg_v[3][i] = 0;
      i = i + 1;
    ) until (i eql 0);
);

if temp_clear[4]
(
    i = 0; next;
    do
    (
      reg_v[4][i] = 0;
      i = i + 1;
    ) until (i eql 0);
);

if temp_clear[5]
(
    i = 0; next;
    do
    (
      reg_v[5][i] = 0;
      i = i + 1;
```

```
        ) until (i eql 0);
    );

    if temp_clear[6]
    (
        i = 0; next;
        do
        (
            reg_v[6][i] = 0;
            i = i + 1;
        ) until (i eql 0);
    );

    if temp_clear[7]
    (
        i = 0; next;
        do
        (
            reg_v[7][i] = 0;
            i = i + 1;
        ) until (i eql 0);
    );
)

when release_lock(clock:PHI_1):=
(
    delay(ND);
    if (lock_granted and fail_latch[lock_set])
    (
        lock_rel = 1;
        lock_granted = 0;
    )
    else if not stall_pipe
    (
        if w_inst_reg_0<ULBIT>
        (
            lock_rel = 1;
            lock_granted = 0;
        )
        else
            lock_rel = 0;;
    )
)

/*************************** Generate the stalls **************************/
when generate_stalls(clock:PHI_1):=
(
    stall_f =  not i_mrdy;
    stall_d =  (case d_stall
                DS_NONE: 0
                DS_R0: not temp_reg0_v
                DS_R1: not temp_reg1_v
                DS_R0R1: not (temp_reg0_v and temp_reg1_v)
                DS_R0R2: not (temp_reg0_v and temp_reg2_v)
                DS_EXEC: not (exec_done_other(d_current_set_0))
                DS_EXECAVAIL: not (free_exec1 or free_exec2 or free_exec3
                    or free_exec4 or free_exec5 or free_exec6 or free_exec7)
                DS_SETAVAIL:  not(
                        case f_current_set_0
                    0: exec_done2
                    1: exec_done3
                    2: exec_done4
                    3: exec_done5
                    4: exec_done6
                    5: exec_done7
                    6: exec_done0
                    7: exec_done1
                    esac)
                DS_QUIT: 1
                esac) or
                (d_inst_reg_0<LBIT> and (not lock_gr) and (not lock_granted));
    stall_m = (not m_done_0) and
                (case m_ctrl_0
                MC_READ,
                MC_PUSH,
                MC_WRITE: not mrdy
                default: 0
                esac);
    stall_w = (not w_done_0) and
                (case w_writereg_0
                WW_GLOBAL: not db_gr
                default: 0
                esac);

    next;

    stall_pipe = stall_f or stall_d or stall_m or stall_w or pref_stall;
    delay(0);
)
```

A3.3. Ideal Multi-port Memory Description

```
/* mem.isp
 One port of ideal multi-port memory
*/
include(../Inc/mem.inc)
include(../Inc/global.inc)
include(../Inc/clock.inc)

port
        maddr<ADDR>   'input,
        mdata<WORD>   'bidirectional:disconnect,
        mrw           'input,
        mrq           'input,
        mrdy (0)      'output:connect,
        clock         'input;

memory
        m[MEMSIZE]<WORD>;

state
        MD<WORD>      (10),  ! memory delay

        ct_rd[0:15]<WORD>,
        ct_wr[0:15]<WORD>;

when mem_port (clock:PHI_0):=
(
        delay(ND);
        if mrq
        (
                if mrw eql READ
                (
                        delay(MD);
                        vconnect(mdata,m[maddr]);
                        mrdy = 1;
                        ct_rd[maddr<15:12>] = ct_rd[maddr<15:12>] + 1;
                        next;

                        wait(clock:PHI_1);

                        delay(MH);
                        disconnect(mdata);
                        mrdy = 0;
                )
                else
                (
                        m[maddr] = mdata;
                        delay(MD);
                        mrdy = 1;
                        ct_wr[maddr<15:12>] = ct_wr[maddr<15:12>] + 1;

                        wait(clock:PHI_1);

                        delay(MH);
                        mrdy = 0;
                )
        );
)
```

A3.4. Update Protocol Snooping Cache Description

```
/* dcacheu.isp

  update protocol snooping cache
*/
include(../Inc/dcacheu.inc)
include(../Inc/global.inc)
include(../Inc/bus.inc)
include(../Inc/clock.inc)

port
    clock          'input,

    maddr<ADDR>    'input,
    mdata<WORD>    'bidirectional:disconnect,
    mrq            'input,
    mrw            'input,
    mpush              'input,
    mrdy (0)       'bidirectional:connect,

    b_maddr<ADDR>        'bidirectional:disconnect,
    b_mdata<WORD>        'bidirectional:disconnect,
    b_code<4:0>      'bidirectional:or:disconnect,

    b_rq (0)       'output:connect,
    b_gr           'input,

    b_busy      (0)    'output:or:connect;


state
    dstate[NUMSETS]<2:0>,
    tags[NUMSETS]<TAGWIDTH>,
    data[NUMSETS][LINESIZE]<WORD>,

    acctimes[0:31]<WORD>,

    p_tag<TAGWIDTH>,
    p_linestate<2:0>,
    p_data<WORD>,

    p_word_ctr<3:0>,
    p_state<4:0> (0),

    b_word_ctr<3:0>,
    b_state<4:0> (0),

    b_maddr_latch<ADDR>,

    p_maddr<ADDR>,
    p_mrw,
    p_mpush;

/*************** processor side ******************************************/

when proc_stall(clock:PHI_0
    ((p_state eql PS_NORMAL) and (b_state neq BS_NORMAL))):=
(
 delay(ND);
 if mrq
 (
      mrdy = 0;
      next;
 );
)

when proc_normal(clock:PHI_0
      ((p_state eql PS_NORMAL) and (b_state eql BS_NORMAL))):=
(
 state buswrite;

 delay(ND); /* wait until bus and processor requests are valid */
 buswrite = ((b_code eql BC_INV) or (b_code eql BC_BROAD)) and
             (tags[b_maddr<SETSELWIDTH>] eql b_maddr<TAGWIDTH>);
 p_maddr = maddr;
 p_mrw = mrw;
 p_mpush = mpush;
 next;

 if mrq and not ((p_maddr<LINEADDR> eql b_maddr<LINEADDR>) and (b_code neq 0))
 (
    p_tag = tags[p_maddr<SETSELWIDTH>];
    p_linestate = dstate[p_maddr<SETSELWIDTH>];
    p_data = data[p_maddr<SETSELWIDTH>][p_maddr<WORDSELWIDTH>];
    next;

    if (p_tag eql p_maddr<TAGWIDTH>)
    (
       /* hit */
       case p_mrw
       READ:
       (
        case p_linestate
        I:
        (
         wait (clock:PHI_1);

         p_state = PS_BUSRQMISS;
        )
        E,M,SM,S:
        (
         vconnect(mdata, p_data);
         mrdy = 1;

         wait(clock:PHI_1);

         delay(CH);
         mrdy = 0;
         disconnect(mdata);
        )
        esac;
       )
       WRITE:
       (
        if p_mpush and (p_maddr<WORDSELWIDTH> eql 0)
        (
         /* push of first word in line */
```

```
case p_linestate
I,SM,S:
(
 wait(clock:PHI_1);

 p_state = PS_BUSRQINV;
)
E,M:
(
    if buswrite
    (
       mrdy = 0;
       wait(clock:PHI_1);
    )
    else
    (
        mrdy = 1;
        data[p_maddr<SETSELWIDTH>][p_maddr<WORDSELWIDTH>] = mdata;
        dstate[p_maddr<SETSELWIDTH>] = M;

        wait(clock:PHI_1);

        delay(CH);
        mrdy = 0;
    )
)
esac;
)
else
(
 /* ordinary write */
 case p_linestate
 I:
 (
  wait (clock:PHI_1);

  p_state = PS_BUSRQMISS;
 )
 S,SM:
 (
  wait (clock:PHI_1);

  p_state = PS_BUSRQBROAD;
 )
 E,M:
 (
    if buswrite
    (
       mrdy = 0;
       wait(clock:PHI_1);
    )
    else
    (
        mrdy = 1;
        data[p_maddr<SETSELWIDTH>][p_maddr<WORDSELWIDTH>] = mdata;
        dstate[p_maddr<SETSELWIDTH>] = M;

        wait(clock:PHI_1);
```

```
                delay(CH);
                mrdy = 0;
             )
          )
          esac;
         )
        )
        esac;
      )
      else
      (
        /* miss */
        case p_linestate
        I,E,S:
        (
         wait (clock:PHI_1);

         if (p_mrw eql WRITE) and (p_mpush and (p_maddr<WORDSELWIDTH> eql 0))
           p_state = PS_BUSRQINV
         else
           p_state = PS_BUSRQMISS;

        )
        M,SM:
        (
         wait(clock:PHI_1);

         p_state = PS_BUSRQWB;
        )
        esac;
      )
    )
  )

when proc_busrqmiss (clock:PHI_0 (p_state eql PS_BUSRQMISS)) :=
(
 /* request bus to ship out request for line */
 b_rq = 1;
 delay(0);


 wait(clock:PHI_1);

 if b_gr
 (
  b_rq = 0;
  b_busy = 1;
  p_state = PS_MISSRQ;
 )
)

when proc_busrqinv (clock:PHI_0 (p_state eql PS_BUSRQINV)) :=
(
 /* request bus to invalidate the line */
 b_rq = 1;
 delay(0);


 wait(clock:PHI_1);
 if b_gr
 (
  b_rq = 0;
  p_state = PS_INV;
 )
)

when proc_busrqbroad (clock:PHI_0 (p_state eql PS_BUSRQBROAD)) :=
(
 /* request bus to broadcast the word */
 b_rq = 1;
 delay(0);


 wait(clock:PHI_1);

 if b_gr
 (
  b_rq = 0;
  p_state = PS_BROAD;
 )
)

when proc_busrqwb (clock:PHI_0 (p_state eql PS_BUSRQWB)) :=
(
 /* request bus to writeback line */
 b_rq = 1;
 delay(0);


 wait(clock:PHI_1);

 if b_gr
 (
  b_rq = 0;
  b_busy = 1;
  p_state = PS_WB;
 )
)

when proc_wb(clock:PHI_0 (p_state eql PS_WB)) :=
(
 vconnect(b_code, BC_WB);
 connect(b_maddr);
 p_word_ctr = 0;
 next;
 b_maddr<WORDSELWIDTH> = p_word_ctr<WORDSELWIDTH>;
 b_maddr<SETSELWIDTH> = p_maddr<SETSELWIDTH>;
 b_maddr<TAGWIDTH> = p_tag;
 delay(0);

 wait(clock:PHI_1);

 p_state = PS_WBDELAY;
)

when proc_wbdelay(clock:PHI_0 (p_state eql PS_WBDELAY)):=
(
 connect(b_mdata);
 wait(clock:PHI_1);

 p_state = PS_WBDATA;
)

when proc_wbdata(clock:PHI_0 (p_state eql PS_WBDATA)):=
(
 b_code = BC_WBDATA;
 b_mdata = data[p_maddr<SETSELWIDTH>][p_word_ctr<WORDSELWIDTH>];
 b_maddr<WORDSELWIDTH> = p_word_ctr<WORDSELWIDTH>;
 delay(ND);
 p_word_ctr = p_word_ctr + 1;
 dstate[p_maddr<SETSELWIDTH>] = I;

 wait(clock:PHI_1);

 if (p_word_ctr eql NUMWORDS)
 (
  if (p_mrw eql READ) or ((p_mrw eql WRITE) and (p_mpush eql 0))
     or ((p_mrw eql WRITE) and (p_mpush eql 1) and (p_maddr<WORDSELWIDTH> neq 0))
  (
   /* read or write */
   p_state = PS_MISSRQ;
  )
  else
  (
   /* push */
   b_busy = 0;
   p_state = PS_INV;
  );

  delay(BH);
  disconnect(b_code);
  disconnect(b_maddr);
  disconnect(b_mdata);
 )
)

when proc_inv(clock:PHI_0 (p_state eql PS_INV)) :=
(
 vconnect(b_code, BC_INV);
 vconnect(b_maddr, p_maddr);
 delay(ND);

 data[p_maddr<SETSELWIDTH>][p_maddr<WORDSELWIDTH>] = mdata;
 tags[p_maddr<SETSELWIDTH>] = p_maddr<TAGWIDTH>;
 dstate[p_maddr<SETSELWIDTH>] = M;
 if (maddr eql p_maddr) mrdy = 1;

 wait(clock:PHI_1);

 p_state = PS_NORMAL;
 delay(BH);
 disconnect(b_code);
 disconnect(b_maddr);
 mrdy = 0;
)

when proc_broad(clock:PHI_0 (p_state eql PS_BROAD)) :=
(
 vconnect(b_code, BC_BROAD);
 vconnect(b_maddr, p_maddr);
 vconnect(b_mdata, mdata);
 delay(ND);

 data[p_maddr<SETSELWIDTH>][p_maddr<WORDSELWIDTH>] = mdata;
 dstate[p_maddr<SETSELWIDTH>] = SM;
 if (maddr eql p_maddr) mrdy = 1;

 wait(clock:PHI_1);

 p_state = PS_NORMAL;
 delay(BH);
 disconnect(b_code);
 disconnect(b_maddr);
 disconnect(b_mdata);
 mrdy = 0;
)

when proc_missrq (clock:PHI_0 (p_state eql PS_MISSRQ)) :=
(
 /* ship out request for line */
 vconnect(b_code, BC_MISSRQ);
 vconnect(b_maddr, p_maddr);
 delay(0);

 wait(clock:PHI_1);

 p_state = PS_MISSWAIT;
 p_word_ctr = 0;
 delay(BH);
 disconnect(b_code);
 disconnect(b_maddr);
)

when proc_misswait(clock:PHI_0 (p_state eql PS_MISSWAIT)) :=
(
 /* wait for line data from either memory or another cache */
 delay(ND); /* wait for bus data to become valid */

 if (b_maddr<LINEADDR> eql p_maddr<LINEADDR>) and
     (b_maddr<WORDSELWIDTH> eql p_word_ctr<WORDSELWIDTH>) and
     ((b_code eql BC_FROM_CACHE) or (b_code eql BC_FROM_MEM))
 (
  data[b_maddr<SETSELWIDTH>][b_maddr<WORDSELWIDTH>] = b_mdata;
  tags[b_maddr<SETSELWIDTH>] = b_maddr<TAGWIDTH>;
  dstate[b_maddr<SETSELWIDTH>] = case b_code
                              BC_FROM_CACHE: S
                              BC_FROM_MEM: E
                              esac;
  p_word_ctr = p_word_ctr + 1;

  wait(clock:PHI_1);

  if (p_word_ctr eql NUMWORDS - 1)
  (
   b_busy = 0;
  )
  else if (p_word_ctr eql NUMWORDS)
  (
   p_state = PS_NORMAL;
  )
 )
)


/************************ bus side ******************************/

when bus_normal(clock:PHI_0 ((b_state eql BS_NORMAL) and
      ((p_state eql PS_NORMAL) or
       (p_state eql PS_BUSRQMISS) or
       (p_state eql PS_BUSRQINV) or
       (p_state eql PS_BUSRQBROAD) or
       (p_state eql PS_BUSRQWB)))) :=
(
 delay(ND);

 case b_code
 BC_INV:
 (
  if (tags[b_maddr<SETSELWIDTH>] eql b_maddr<TAGWIDTH>)
   dstate[b_maddr<SETSELWIDTH>] = I;
 )
 BC_BROAD:
 (
  if (tags[b_maddr<SETSELWIDTH>] eql b_maddr<TAGWIDTH>)
  (
   if (dstate[b_maddr<SETSELWIDTH>] eql S) or
      (dstate[b_maddr<SETSELWIDTH>] eql SM)
   (
     data[b_maddr<SETSELWIDTH>][b_maddr<WORDSELWIDTH>] = b_mdata;
     dstate[b_maddr<SETSELWIDTH>] = S;
   )
   else if (dstate[b_maddr<SETSELWIDTH>] eql E) or
        (dstate[b_maddr<SETSELWIDTH>] eql M)
   (
     notify(UNEXPECTED_BROADCAST);
   )
  )
 )
 BC_MISSRQ:
 (
  if (tags[b_maddr<SETSELWIDTH>] eql b_maddr<TAGWIDTH>) and
    (dstate[b_maddr<SETSELWIDTH>] neq I)
  (
   b_word_ctr = 0;
   b_maddr_latch = b_maddr;

   wait(clock:PHI_1);

   b_state = BS_MISS;
  )
 )
 esac
)

when bus_miss(clock:PHI_0 (b_state eql BS_MISS)) :=
(
 vconnect(b_code, BC_FROM_CACHE);
 vconnect(b_mdata, data[b_maddr_latch<SETSELWIDTH>][b_word_ctr<WORDSELWIDTH>]);
 connect(b_maddr);
 next;
 b_maddr<LINEADDR> = b_maddr_latch<LINEADDR>;
 b_maddr<WORDSELWIDTH> = b_word_ctr<WORDSELWIDTH>;
 delay(ND);
 b_word_ctr = b_word_ctr + 1;
 case dstate[b_maddr_latch<SETSELWIDTH>]
 M,SM: dstate[b_maddr_latch<SETSELWIDTH>] = SM
 E,S: dstate[b_maddr_latch<SETSELWIDTH>] = S
 esac;

 wait(clock:PHI_1);

 if (b_word_ctr eql NUMWORDS)
  b_state = BS_NORMAL;

 delay(BH);
 disconnect(b_code);
 disconnect(b_mdata);
 disconnect(b_maddr);
)

when access_time_monitor(mrq:lead):=
(
  state acctime<WORD>;

  wait(clock:PHI_1);

  acctime = 1;
  next;
  while(mrq and not mrdy)
  (
     wait(clock:PHI_1);

     acctime = acctime + 1;
  );
  next;

  if (acctime gtr 30)
  (
     acctimes[31] = acctimes[31] + 1;
  )
  else
  (
     acctimes[acctime] = acctimes[acctime] + 1;
  );
)
```

A3.5. Distribution Bus and Lock Arbitration

```
/*      arbit.isp
        arbitration unit for the FPPM dist_bus and lock_rq
*/


include(../Inc/clock.inc)
include(../Inc/global.inc)

port
        clock           'input,

        db_rq0              'input,
        db_gr0 (0)  'output:connect,
        db_rq1              'input,
        db_gr1 (0)  'output:connect,
        db_rq2              'input,
        db_gr2 (0)  'output:connect,
        db_rq3              'input,
        db_gr3 (0)  'output:connect,
        db_rq4              'input,
        db_gr4 (0)  'output:connect,
        db_rq5              'input,
        db_gr5 (0)  'output:connect,
        db_rq6              'input,
        db_gr6 (0)  'output:connect,
        db_rq7              'input,
        db_gr7 (0)  'output:connect,

        lock_rq0    'input,
        lock_gr0 (0)'output:connect,
        lock_rq1    'input,
        lock_gr1 (0)'output:connect,
        lock_rq2    'input,
        lock_gr2 (0)'output:connect,
        lock_rq3    'input,
        lock_gr3 (0)'output:connect,
        lock_rq4    'input,
        lock_gr4 (0)'output:connect,
        lock_rq5    'input,
        lock_gr5 (0)'output:connect,
        lock_rq6    'input,
        lock_gr6 (0)'output:connect,
        lock_rq7    'input,
        lock_gr7 (0)'output:connect,

        lock_rel0   'input,
        lock_rel1   'input,
        lock_rel2   'input,
        lock_rel3   'input,
        lock_rel4   'input,
        lock_rel5   'input,
        lock_rel6   'input,
        lock_rel7   'input,

        pref_stall  'input;

state
```

```
        last_lock<2:0> (0),
        last_dist<2:0> (0),
        lock_grant (0),
        db_rq_ct[0:7]<WORD>,
        lock_rq_ct[0:7]<WORD>;

when arbit(clock: PHI_0):=
(
        state temp_gr[0:7], i<2:0>, dist_grant;

        delay(ND); /* wait for lock_rq, db_rq */

    if lock_grant     /* check if lock has been released */
    (
        if (case last_lock
           0: lock_rel7
           1: lock_rel0
           2: lock_rel1
           3: lock_rel2
           4: lock_rel3
           5: lock_rel4
           6: lock_rel5
           7: lock_rel6
           esac)
        (
           lock_grant = 0;
        )
    );

        /* initialize temp_gr */
        i = 0;
        next;
        do
        (
           temp_gr[i] = 0;
           i = i + 1;
        ) until ( i eql 0);
        next;


    if not lock_grant
    (
        /* find the first lock_rq in round_robin order */
        i = last_lock;
        next;

        do
        (
           if (case i
               0: lock_rq0
               1: lock_rq1
               2: lock_rq2
               3: lock_rq3
               4: lock_rq4
               5: lock_rq5
               6: lock_rq6
               7: lock_rq7
               esac)
           (
                lock_grant = 1;
                temp_gr[i] = 1;
             );
             i = i + 1;
          ) until ((i eql last_lock) or lock_grant);
          last_lock = i;
       );
       next;

          /* ship out lock_gr signals */
          lock_gr0 = temp_gr[0];
          lock_gr1 = temp_gr[1];
          lock_gr2 = temp_gr[2];
          lock_gr3 = temp_gr[3];
          lock_gr4 = temp_gr[4];
          lock_gr5 = temp_gr[5];
          lock_gr6 = temp_gr[6];
          lock_gr7 = temp_gr[7];

          /* initialize temp_gr */
          i = 0;
          next;
          do
          (
             temp_gr[i] = 0;
             i = i + 1;
          ) until ( i eql 0);
          dist_grant = 0;
          next;

          /* if lock was granted, check if the same unit has a db_rq */
          if lock_grant
          (
                case last_lock
                0: temp_gr[7] = db_rq7
                1: temp_gr[0] = db_rq0
                2: temp_gr[1] = db_rq1
                3: temp_gr[2] = db_rq2
                4: temp_gr[3] = db_rq3
                5: temp_gr[4] = db_rq4
                6: temp_gr[5] = db_rq5
                7: temp_gr[6] = db_rq6
                esac;
                next;
                dist_grant = (temp_gr[0] or temp_gr[1] or temp_gr[2] or
                        temp_gr[3] or temp_gr[4] or temp_gr[5] or
                        temp_gr[6] or temp_gr[7]);
                next;
                if dist_grant
                (
                   last_dist = last_lock;
                );
          );

          if not dist_grant
          (
                i = last_dist;
                next;
                do
                (
                  if (case i
                      0: db_rq0
                      1: db_rq1
                      2: db_rq2
                      3: db_rq3
                      4: db_rq4
                      5: db_rq5
                      6: db_rq6
                      7: db_rq7
                      esac)
                  (
                      dist_grant = 1;
                      temp_gr[i] = 1;
                  );
                  i = i + 1;
                ) until ((i eql last_dist) or dist_grant);
                last_dist = i;
                next;
        );

        /* ship out db_gr signals */
        db_gr0 = temp_gr[0];
        db_gr1 = temp_gr[1];
        db_gr2 = temp_gr[2];
        db_gr3 = temp_gr[3];
        db_gr4 = temp_gr[4];
        db_gr5 = temp_gr[5];
        db_gr6 = temp_gr[6];
        db_gr7 = temp_gr[7];
)

when db_arb_monitor(clock:PHI_0):=
(
  state temp<2:0>;

   delay(ND);
   temp  = (db_rq0 ext 4) +
           (db_rq1 ext 4) +
           (db_rq2 ext 4) +
           (db_rq3 ext 4) +
           (db_rq4 ext 4) +
           (db_rq5 ext 4) +
           (db_rq6 ext 4) +
           (db_rq7 ext 4);
   next;
   db_rq_ct[temp] = db_rq_ct[temp] + 1;
)

when lock_arb_monitor(clock:PHI_0):=
(
  state temp<2:0>;

   delay(ND);
   temp  = (lock_rq0 ext 4) +
           (lock_rq1 ext 4) +
           (lock_rq2 ext 4) +
           (lock_rq3 ext 4) +
           (lock_rq4 ext 4) +
                (lock_rq5 ext 4) +
                (lock_rq6 ext 4) +
                (lock_rq7 ext 4);
        next;
        lock_rq_ct[temp] = lock_rq_ct[temp] + 1;
    )
```

A3.6. Example Ecology File

```
/*
  Ecology file with
  1 Main Processor
  4 Slave Processors
  Update protocol data caches
*/

/*
 first define signals common to all configs
*/
signal
      clock<1>,

      i_maddr0<28>,
      i_mdata0<32>,
      i_mrw0<1>,
      i_mrq0<1>,
      i_mrdy0<1>,

      maddr0<28>,
      mdata0<32>,
      mrw0<1>,
      mpush0<1>,
      mrq0<1>,
      mrdy0<1>,

      free_exec1<1>,
      free_exec2<1>,
      free_exec3<1>,
      free_exec4<1>,
      free_exec5<1>,
      free_exec6<1>,
      free_exec7<1>,

      task_bus<37>,

      pref_stall<1>,
      pref_set<3>,

      dist_bus<42>,

      db_rq0<1>,
      db_rq1<1>,
      db_rq2<1>,
      db_rq3<1>,
      db_rq4<1>,
      db_rq5<1>,
      db_rq6<1>,
      db_rq7<1>,

      db_gr0<1>,
      db_gr1<1>,
      db_gr2<1>,
      db_gr3<1>,
      db_gr4<1>,
      db_gr5<1>,
      db_gr6<1>,
        db_gr7<1>,

        lock_rq0<1>,
        lock_rq1<1>,
        lock_rq2<1>,
        lock_rq3<1>,
        lock_rq4<1>,
        lock_rq5<1>,
        lock_rq6<1>,
        lock_rq7<1>,

        lock_gr0<1>,
        lock_gr1<1>,
        lock_gr2<1>,
        lock_gr3<1>,
        lock_gr4<1>,
        lock_gr5<1>,
        lock_gr6<1>,
        lock_gr7<1>,

        lock_rel0<1>,
        lock_rel1<1>,
        lock_rel2<1>,
        lock_rel3<1>,
        lock_rel4<1>,
        lock_rel5<1>,
        lock_rel6<1>,
        lock_rel7<1>,


        exec_done0<1>,
        exec_done1<1>,
        exec_done2<1>,
        exec_done3<1>,
        exec_done4<1>,
        exec_done5<1>,
        exec_done6<1>,
        exec_done7<1>,

        clear_sig0<1>,
        clear_sig1<1>,
        clear_sig2<1>,
        clear_sig3<1>,
        clear_sig4<1>,
        clear_sig5<1>,
        clear_sig6<1>,
        clear_sig7<1>,

        fail_sig0<1>,
        fail_sig1<1>,
        fail_sig2<1>,
        fail_sig3<1>,
        fail_sig4<1>,
        fail_sig5<1>,
        fail_sig6<1>,
        fail_sig7<1>;

/*
        signals for shared memory bus
```

```
*/
signal
      /* shared memory bus */

      b_maddr<28>,
      b_mdata<32>,
      b_code<5>,
      b_busy<1>,

      /* arbitration signals for bus */

      b_rq0<1>,
      b_gr0<1>,

      b_rq1<1>,
      b_gr1<1>,

      b_rq2<1>,
      b_gr2<1>,

      b_rq3<1>,
      b_gr3<1>,

      b_rq4<1>,
      b_gr4<1>,

      b_rq5<1>,
      b_gr5<1>,

      b_rq6<1>,
      b_gr6<1>,

      b_rq7<1>,
      b_gr7<1>,

      b_rq8<1>,
      b_gr8<1>,

      b_rq9<1>,
      b_gr9<1>,

      b_rqa<1>,
      b_gra<1>,

      b_rqb<1>,
      b_grb<1>,

      b_rqc<1>,
      b_grc<1>,

      b_rqd<1>,
      b_grd<1>,

      b_rqe<1>,
      b_gre<1>,

      b_rqf<1>,
      b_grf<1>,
        b_rq10<1>,
        b_gr10<1>;

/*
 Extra signals for Slave Processors
*/
signal
        i_maddr1<28>,
        i_mdata1<32>,
        i_mrw1<1>,
        i_mrq1<1>,
        i_mrdy1<1>,

        maddr1<28>,
        mdata1<32>,
        mrw1<1>,
        mpush1<1>,
        mrq1<1>,
        mrdy1<1>;

signal
        i_maddr2<28>,
        i_mdata2<32>,
        i_mrw2<1>,
        i_mrq2<1>,
        i_mrdy2<1>,

        maddr2<28>,
        mdata2<32>,
        mrw2<1>,
        mpush2<1>,
        mrq2<1>,
        mrdy2<1>;

signal
        i_maddr3<28>,
        i_mdata3<32>,
        i_mrw3<1>,
        i_mrq3<1>,
        i_mrdy3<1>,

        maddr3<28>,
        mdata3<32>,
        mrw3<1>,
        mpush3<1>,
        mrq3<1>,
        mrdy3<1>;

signal
        i_maddr4<28>,
        i_mdata4<32>,
        i_mrw4<1>,
        i_mrq4<1>,
        i_mrdy4<1>,

        maddr4<28>,
        mdata4<32>,
        mrw4<1>,
        mpush4<1>,
        mrq4<1>,
        mrdy4<1>;

/*
 the processor declaration for the minimum FPPM with dcache
*/
processor clock = clock.sim'isp;
time delay = 1;
connections
        clock = clock;

processor mainproc = mainproc.sim'isp;
time delay = 1;
initial uinst_mem = mainproc.out'simulated;
connections
        clock = clock,

        i_maddr = i_maddr0,
        i_mdata = i_mdata0,
        i_mrq = i_mrq0,
        i_mrw = i_mrw0,
        i_mrdy = i_mrdy0,

        maddr = maddr0,
        mdata = mdata0,
        mrq = mrq0,
        mrdy = mrdy0,
        mrw = mrw0,
        mpush = mpush0,

        free_exec1 = free_exec1,
        free_exec2 = free_exec2,
        free_exec3 = free_exec3,
        free_exec4 = free_exec4,
        free_exec5 = free_exec5,
        free_exec6 = free_exec6,
        free_exec7 = free_exec7,

        task_bus = task_bus,
        pref_stall = pref_stall,
        pref_set = pref_set,

        dist_bus = dist_bus,
        db_rq = db_rq0,
        db_gr = db_gr0,

        lock_rel = lock_rel0,
        lock_rq = lock_rq0,
        lock_gr = lock_gr0,

        fail_sig0 = fail_sig0,
        fail_sig1 = fail_sig1,
        fail_sig2 = fail_sig2,
        fail_sig3 = fail_sig3,
        fail_sig4 = fail_sig4,
        fail_sig5 = fail_sig5,
        fail_sig6 = fail_sig6,
        fail_sig7 = fail_sig7,
        clear_sig0 = clear_sig0,
        clear_sig1 = clear_sig1,
        clear_sig2 = clear_sig2,
        clear_sig3 = clear_sig3,
        clear_sig4 = clear_sig4,
        clear_sig5 = clear_sig5,
        clear_sig6 = clear_sig6,
        clear_sig7 = clear_sig7,

        exec_done0 = exec_done0,
        exec_done1 = exec_done1,
        exec_done2 = exec_done2,
        exec_done3 = exec_done3,
        exec_done4 = exec_done4,
        exec_done5 = exec_done5,
        exec_done6 = exec_done6,
        exec_done7 = exec_done7;

processor i_mem0 = mem.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

        clock = clock,

        maddr = i_maddr0,
        mdata = i_mdata0,
        mrw = i_mrw0,
        mrq = i_mrq0,
        mrdy = i_mrdy0;

processor dcache0 = dcacheu.sim'isp;
time delay = 1;
connections

        clock = clock,

        maddr = maddr0,
        mdata = mdata0,
        mrw = mrw0,
        mpush = mpush0,
        mrq = mrq0,
        mrdy = mrdy0,

        b_maddr = b_maddr,
        b_mdata = b_mdata,
        b_code = b_code,
        b_busy = b_busy,

        b_rq = b_rq0,
        b_gr = b_gr0;

processor arbit = arbit.sim'isp;
connections
        clock = clock,

        pref_stall = pref_stall,

        db_rq0 = db_rq0,
          db_gr0 = db_gr0,
          db_rq1 = db_rq1,
          db_gr1 = db_gr1,
          db_rq2 = db_rq2,
          db_gr2 = db_gr2,
          db_rq3 = db_rq3,
          db_gr3 = db_gr3,
          db_rq4 = db_rq4,
          db_gr4 = db_gr4,
          db_rq5 = db_rq5,
          db_gr5 = db_gr5,
          db_rq6 = db_rq6,
          db_gr6 = db_gr6,
          db_rq7 = db_rq7,
          db_gr7 = db_gr7,

          lock_rel0 = lock_rel0,
          lock_rel1 = lock_rel1,
          lock_rel2 = lock_rel2,
          lock_rel3 = lock_rel3,
          lock_rel4 = lock_rel4,
          lock_rel5 = lock_rel5,
          lock_rel6 = lock_rel6,
          lock_rel7 = lock_rel7,

          lock_rq0 = lock_rq0,
          lock_gr0 = lock_gr0,
          lock_rq1 = lock_rq1,
          lock_gr1 = lock_gr1,
          lock_rq2 = lock_rq2,
          lock_gr2 = lock_gr2,
          lock_rq3 = lock_rq3,
          lock_gr3 = lock_gr3,
          lock_rq4 = lock_rq4,
          lock_gr4 = lock_gr4,
          lock_rq5 = lock_rq5,
          lock_gr5 = lock_gr5,
          lock_rq6 = lock_rq6,
          lock_gr6 = lock_gr6,
          lock_rq7 = lock_rq7,
          lock_gr7 = lock_gr7;
/* mbusproc.t
          processor declarations for memory bus
*/

/* shared memory */
processor dmemu = dmemu.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

          clock = clock,

          b_maddr = b_maddr,
          b_mdata = b_mdata,
          b_code = b_code,
          b_busy = b_busy,

          b_rq = b_rq10,
          b_gr = b_gr10;


 /* bus arbiter for shared bus */
processor busarb = shbusarb.sim'isp;
time delay = 1;
connections

          clock = clock,

          b_busy = b_busy,
          b_code = b_code,

          b_rq0 = b_rq0,
          b_rq1 = b_rq1,
          b_rq2 = b_rq2,
          b_rq3 = b_rq3,
          b_rq4 = b_rq4,
          b_rq5 = b_rq5,
          b_rq6 = b_rq6,
          b_rq7 = b_rq7,
          b_rq8 = b_rq8,
          b_rq9 = b_rq9,
          b_rqa = b_rqa,
          b_rqb = b_rqb,
          b_rqc = b_rqc,
          b_rqd = b_rqd,
          b_rqe = b_rqe,
          b_rqf = b_rqf,
          b_rq10 = b_rq10,

          b_gr0 = b_gr0,
          b_gr1 = b_gr1,
          b_gr2 = b_gr2,
          b_gr3 = b_gr3,
          b_gr4 = b_gr4,
          b_gr5 = b_gr5,
          b_gr6 = b_gr6,
          b_gr7 = b_gr7,
          b_gr8 = b_gr8,
          b_gr9 = b_gr9,
          b_gra = b_gra,
          b_grb = b_grb,
          b_grc = b_grc,
          b_grd = b_grd,
          b_gre = b_gre,
          b_grf = b_grf,
          b_gr10 = b_gr10;

processor slave1 = slave.sim'isp;
time delay = 1;
initial uinst_mem = slave.out'simulated;
connections
          clock = clock,

          i_maddr = i_maddr1,
          i_mdata = i_mdata1,
          i_mrq = i_mrq1,
          i_mrw = i_mrw1,
        i_mrdy = i_mrdy1,

        maddr = maddr1,
        mdata = mdata1,
        mrq = mrq1,
        mrw = mrw1,
        mpush = mpush1,
        mrdy = mrdy1,

        free_exec = free_exec1,

        task_bus = task_bus,

        dist_bus = dist_bus,
        db_rq = db_rq1,
        db_gr = db_gr1,

        lock_rel = lock_rel1,
        lock_rq = lock_rq1,
        lock_gr = lock_gr1,

        pref_stall = pref_stall,
        pref_set = pref_set,

        fail_sig0 = fail_sig0,
        fail_sig1 = fail_sig1,
        fail_sig2 = fail_sig2,
        fail_sig3 = fail_sig3,
        fail_sig4 = fail_sig4,
        fail_sig5 = fail_sig5,
        fail_sig6 = fail_sig6,
        fail_sig7 = fail_sig7,

        clear_sig0 = clear_sig0,
        clear_sig1 = clear_sig1,
        clear_sig2 = clear_sig2,
        clear_sig3 = clear_sig3,
        clear_sig4 = clear_sig4,
        clear_sig5 = clear_sig5,
        clear_sig6 = clear_sig6,
        clear_sig7 = clear_sig7,

        exec_done0 = exec_done0,
        exec_done1 = exec_done1,
        exec_done2 = exec_done2,
        exec_done3 = exec_done3,
        exec_done4 = exec_done4,
        exec_done5 = exec_done5,
        exec_done6 = exec_done6,
        exec_done7 = exec_done7;

processor i_mem1 = mem.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

        clock = clock,

        maddr = i_maddr1,
        mdata = i_mdata1,
        mrw = i_mrw1,
        mrq = i_mrq1,
        mrdy = i_mrdy1;

processor dcache1 = dcacheu.sim'isp;
time delay = 1;
connections

        clock = clock,

        maddr = maddr1,
        mdata = mdata1,
        mrw = mrw1,
        mpush = mpush1,
        mrq = mrq1,
        mrdy = mrdy1,

        b_maddr = b_maddr,
        b_mdata = b_mdata,
        b_code = b_code,
        b_busy = b_busy,

        b_rq = b_rq1,
        b_gr = b_gr1;

processor slave2 = slave.sim'isp;
time delay = 1;
initial uinst_mem = slave.out'simulated;
connections
        clock = clock,

        i_maddr = i_maddr2,
        i_mdata = i_mdata2,
        i_mrq = i_mrq2,
        i_mrw = i_mrw2,
        i_mrdy = i_mrdy2,

        maddr = maddr2,
        mdata = mdata2,
        mrq = mrq2,
        mrw = mrw2,
        mpush = mpush2,
        mrdy = mrdy2,

        free_exec = free_exec2,

        task_bus = task_bus,

        dist_bus = dist_bus,
        db_rq = db_rq2,
        db_gr = db_gr2,

        lock_rel = lock_rel2,
        lock_rq = lock_rq2,
        lock_gr = lock_gr2,

        pref_stall = pref_stall,
        pref_set = pref_set,
        fail_sig0 = fail_sig0,
        fail_sig1 = fail_sig1,
        fail_sig2 = fail_sig2,
        fail_sig3 = fail_sig3,
        fail_sig4 = fail_sig4,
        fail_sig5 = fail_sig5,
        fail_sig6 = fail_sig6,
        fail_sig7 = fail_sig7,

        clear_sig0 = clear_sig0,
        clear_sig1 = clear_sig1,
        clear_sig2 = clear_sig2,
        clear_sig3 = clear_sig3,
        clear_sig4 = clear_sig4,
        clear_sig5 = clear_sig5,
        clear_sig6 = clear_sig6,
        clear_sig7 = clear_sig7,

        exec_done0 = exec_done0,
        exec_done1 = exec_done1,
        exec_done2 = exec_done2,
        exec_done3 = exec_done3,
        exec_done4 = exec_done4,
        exec_done5 = exec_done5,
        exec_done6 = exec_done6,
        exec_done7 = exec_done7;

processor i_mem2 = mem.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

        clock = clock,

        maddr = i_maddr2,
        mdata = i_mdata2,
        mrw = i_mrw2,
        mrq = i_mrq2,
        mrdy = i_mrdy2;

processor dcache2 = dcacheu.sim'isp;
time delay = 1;
connections

        clock = clock,

        maddr = maddr2,
        mdata = mdata2,
        mrw = mrw2,
        mpush = mpush2,
        mrq = mrq2,
        mrdy = mrdy2,

        b_maddr = b_maddr,
        b_mdata = b_mdata,
        b_code = b_code,
        b_busy = b_busy,

        b_rq = b_rq2,
          b_gr = b_gr2;

processor slave3 = slave.sim'isp;
time delay = 1;
initial uinst_mem = slave.out'simulated;
connections
          clock = clock,

          i_maddr = i_maddr3,
          i_mdata = i_mdata3,
          i_mrq = i_mrq3,
          i_mrw = i_mrw3,
          i_mrdy = i_mrdy3,

          maddr = maddr3,
          mdata = mdata3,
          mrq = mrq3,
          mrw = mrw3,
          mpush = mpush3,
          mrdy = mrdy3,

          free_exec = free_exec3,

          task_bus = task_bus,

          dist_bus = dist_bus,
          db_rq = db_rq3,
          db_gr = db_gr3,

          lock_rel = lock_rel3,
          lock_rq = lock_rq3,
          lock_gr = lock_gr3,

          pref_stall = pref_stall,
          pref_set = pref_set,

          fail_sig0 = fail_sig0,
          fail_sig1 = fail_sig1,
          fail_sig2 = fail_sig2,
          fail_sig3 = fail_sig3,
          fail_sig4 = fail_sig4,
          fail_sig5 = fail_sig5,
          fail_sig6 = fail_sig6,
          fail_sig7 = fail_sig7,

          clear_sig0 = clear_sig0,
          clear_sig1 = clear_sig1,
          clear_sig2 = clear_sig2,
          clear_sig3 = clear_sig3,
          clear_sig4 = clear_sig4,
          clear_sig5 = clear_sig5,
          clear_sig6 = clear_sig6,
          clear_sig7 = clear_sig7,

          exec_done0 = exec_done0,
          exec_done1 = exec_done1,
          exec_done2 = exec_done2,
          exec_done3 = exec_done3,
          exec_done4 = exec_done4,
        exec_done5 = exec_done5,
        exec_done6 = exec_done6,
        exec_done7 = exec_done7;

processor i_mem3 = mem.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

        clock = clock,

        maddr = i_maddr3,
        mdata = i_mdata3,
        mrw = i_mrw3,
        mrq = i_mrq3,
        mrdy = i_mrdy3;

processor dcache3 = dcacheu.sim'isp;
time delay = 1;
connections

        clock = clock,

        maddr = maddr3,
        mdata = mdata3,
        mrw = mrw3,
        mpush = mpush3,
        mrq = mrq3,
        mrdy = mrdy3,

        b_maddr = b_maddr,
        b_mdata = b_mdata,
        b_code = b_code,
        b_busy = b_busy,

        b_rq = b_rq3,
        b_gr = b_gr3;

processor slave4 = slave.sim'isp;
time delay = 1;
initial uinst_mem = slave.out'simulated;
connections
        clock = clock,

        i_maddr = i_maddr4,
        i_mdata = i_mdata4,
        i_mrq = i_mrq4,
        i_mrw = i_mrw4,
        i_mrdy = i_mrdy4,

        maddr = maddr4,
        mdata = mdata4,
        mrq = mrq4,
        mrw = mrw4,
        mpush = mpush4,
        mrdy = mrdy4,

        free_exec = free_exec4,
        task_bus = task_bus,

        dist_bus = dist_bus,
        db_rq = db_rq4,
        db_gr = db_gr4,

        lock_rel = lock_rel4,
        lock_rq = lock_rq4,
        lock_gr = lock_gr4,

        pref_stall = pref_stall,
        pref_set = pref_set,

        fail_sig0 = fail_sig0,
        fail_sig1 = fail_sig1,
        fail_sig2 = fail_sig2,
        fail_sig3 = fail_sig3,
        fail_sig4 = fail_sig4,
        fail_sig5 = fail_sig5,
        fail_sig6 = fail_sig6,
        fail_sig7 = fail_sig7,

        clear_sig0 = clear_sig0,
        clear_sig1 = clear_sig1,
        clear_sig2 = clear_sig2,
        clear_sig3 = clear_sig3,
        clear_sig4 = clear_sig4,
        clear_sig5 = clear_sig5,
        clear_sig6 = clear_sig6,
        clear_sig7 = clear_sig7,

        exec_done0 = exec_done0,
        exec_done1 = exec_done1,
        exec_done2 = exec_done2,
        exec_done3 = exec_done3,
        exec_done4 = exec_done4,
        exec_done5 = exec_done5,
        exec_done6 = exec_done6,
        exec_done7 = exec_done7;

processor i_mem4 = mem.sim'isp;
time delay = 1;
initial m = program.out'simulated;
connections

        clock = clock,

        maddr = i_maddr4,
        mdata = i_mdata4,
        mrw = i_mrw4,
        mrq = i_mrq4,
        mrdy = i_mrdy4;

processor dcache4 = dcacheu.sim'isp;
time delay = 1;
connections

        clock = clock,
maddr = maddr4,
mdata = mdata4,
mrw = mrw4,
mpush = mpush4,
mrq = mrq4,
mrdy = mrdy4,

b_maddr = b_maddr,
b_mdata = b_mdata,
b_code = b_code,
b_busy = b_busy,

b_rq = b_rq4,
b_gr = b_gr4;
```

A3.7. Control Words for Main Processor

```
% mainproc.m

%

include ../Prefmic/mainproc.format  $
include ../Prefmic/mainproc.uinst   $

begin
.   =     uQUIT      $
    d_stall          =     DS_QUIT  $

.   =     uEXEC      $
    d_branchlength   =     DB_LONGLAB;
    d_stall          =     DS_EXECAVAIL;
    d_dispatch  =    DD_EXEC $


.   =     uNEW       $
    d_regset    =    DR_NEW;
    d_stall          =     DS_SETAVAIL  $

.   =     uOLD       $
    d_regset    =    DR_OLD   $

.   =     uFAIL      $
    d_fail      =    DL_FAIL  $

.   =     uCLEARFAIL    $
    d_fail      =    DL_CLEAR       $

.   =     uCLEARCP      $
    d_fail      =    DL_CLEARCP   $

.   =     uWAITEXEC     $      ! wait till previous set's execs done
    d_stall          =     DS_EXEC  $


.   =     uARITH     $      ! r1 <- r0 scode r2, r1 is local
    d_stall          =     DS_R0R2;
    a_bopsrc    =    AB_R2;
    a_alucode   =    AA_SCODE;
    a_bypass    =    AP_YES;
    m_bypass    =    MP_YES;
    w_writereg  =    WW_LOCAL      $

.   =     uARITHG  $      ! r1 <- r0 scode r2, r1 is global
    d_stall          =     DS_R0R2;
    a_bopsrc    =    AB_R2;
    a_alucode   =    AA_SCODE;
    w_writereg  =    WW_GLOBAL   $

.   =     uARITHOFF     $      ! r1 <- r0 scode off, r1 is local
    d_stall          =     DS_R0;
    a_alucode   =    AA_SCODE;
    a_bypass    =    AP_YES;
    m_bypass    =    MP_YES;
    w_writereg  =    WW_LOCAL      $
```

```
.   =     uARITHOFFG    $     ! r1 <- r0 scode off, r1 is global
    d_stall          =     DS_R0;
    a_alucode   =    AA_SCODE;
    w_writereg =     WW_GLOBAL   $

.   =     uMOVETAG      $     ! r1 <- tag^(r0v + off), r1 is local
    d_stall          =     DS_R0;
    a_tagsrc    =    AT_ITAG;
    a_bypass    =    AP_YES;
    m_bypass    =    MP_YES;
    w_writereg =     WW_LOCAL    $

.   =     uMOVETAGG     $     ! r1 <- tag^(r0v + off), r1 is global
    d_stall          =     DS_R0;
    a_tagsrc    =    AT_ITAG;
    w_writereg =     WW_GLOBAL   $

.   =     uPUSH         $     ! m(r1) <- r0 + off, r1 <- r1+1,r1 local
    d_stall          =     DS_R0R1;
    a_marsrc    =    AM_R1;
    a_result    =    AR_INC;
    a_bypass    =    AP_YES;
    m_bypass    =    MP_YES;
    m_ctrl           =     MC_PUSH;
    w_writereg =     WW_LOCAL    $

.   =     uPUSHG $            ! m(r1) <- r0 + off, r1 <-r1+1,r1 global
    d_stall          =     DS_R0R1;
    a_marsrc    =    AM_R1;
    a_result    =    AR_INC;
    m_ctrl           =     MC_PUSH;
    w_writereg =     WW_GLOBAL   $

.   =     uPUSHTAG      $     ! m(r1) <- tag^(r0v + off), r1 <- r1+1,r1 local
    d_stall          =     DS_R0R1;
    a_tagsrc    =    AT_ITAG;
    a_marsrc    =    AM_R1;
    a_result    =    AR_INC;
    a_bypass    =    AP_YES;
    m_bypass    =    MP_YES;
    m_ctrl           =     MC_PUSH;
    w_writereg =     WW_LOCAL    $

.   =     uPUSHTAGG $   ! m(r1) <- tag^(r0v + off), r1 <-r1+1,r1 global
    d_stall          =     DS_R0R1;
    a_tagsrc    =    AT_ITAG;
    a_marsrc    =    AM_R1;
    a_result    =    AR_INC;
    m_ctrl           =     MC_PUSH;
    w_writereg =     WW_GLOBAL   $

.   =     uLOAD     $     ! r1 <- m(r0 + off), r1 local
    d_stall          =     DS_R0;
    m_ctrl           =     MC_READ;
    m_bypass    =    MP_YES;
    w_writereg =     WW_LOCAL    $

.   =     uLOADG    $     ! r1 <- m(r0 + off), r1 global
    d_stall          =     DS_R0;
```

```
          m_ctrl          =    MC_READ;
          w_writereg =    WW_GLOBAL   $

     .    =    uSTORE    $    ! m(r0 + off) <- r1
          d_stall         =    DS_R0R1;
          a_mdrsrc   =    AD_R1;
          m_ctrl          =    MC_WRITE      $

     .    =    uBR  $    ! fpc <- pc + longlab
          d_branchlength  =    DB_LONGLAB;
          d_fpcctl   =    DF_BRANCH   $

     .    =    uSBR $    ! fpc <- pc + longlab, squash
          d_branchlength  =    DB_LONGLAB;
          d_fpcctl   =    DF_BRANCH;
          d_squash   =    DQ_YES   $

     .    =    uBRIND    $    ! fpc <- reg1
          d_fpcctl   =    DF_REG1;
          d_stall         =    DS_R1     $

     .    =    uSBRIND   $    ! fpc <- reg1,squash
          d_fpcctl   =    DF_REG1;
          d_stall         =    DS_R1;
          d_squash   =    DQ_YES   $

     .    =    uBRAL     $    ! fpc <- pc + lab, r1 <- pc+1, r1 local
          d_branchlength  =    DB_LAB;
          d_fpcctl   =    DF_BRANCH;
          a_incsrc   =    AI_PC;
          a_result   =    AR_INC;
          a_bypass   =    AP_YES;
          m_bypass   =    MP_YES;
          w_writereg =    WW_LOCAL     $

     .    =    uBRALG    $    ! fpc <- pc + lab, r1 <- pc+1, r1 global
          d_branchlength  =    DB_LAB;
          d_fpcctl   =    DF_BRANCH;
          a_incsrc   =    AI_PC;
          a_result   =    AR_INC;
          w_writereg =    WW_GLOBAL   $

     .    =    uSBRAL    $    ! fpc <- pc + lab, r1 <-pc+1, r1 local,squash
          d_branchlength  =    DB_LAB;
          d_fpcctl   =    DF_BRANCH;
          d_squash   =    DQ_YES;
          a_incsrc   =    AI_PC;
          a_result   =    AR_INC;
          a_bypass   =    AP_YES;
          m_bypass   =    MP_YES;
          w_writereg =    WW_LOCAL     $

     .    =    uSBRALG $    ! fpc <- pc + lab, r1 <-pc + 1,r1 global,squash
          d_branchlength  =    DB_LAB;
          d_fpcctl   =    DF_BRANCH;
          d_squash   =    DQ_YES;
          a_incsrc   =    AI_PC;
          a_result   =    AR_INC;
          w_writereg =    WW_GLOBAL   $
```

```
.   =     uBRCOND $       !if cond(r1) fpc <- pc + lab
d_fpcctl      =     DF_COND;
d_stall              =     DS_R1;
d_cond               =     DC_COND $

.   =     uBRNCOND      $      !if ncond(r1) fpc <- pc + lab
d_fpcctl      =     DF_COND;
d_stall              =     DS_R1;
d_cond               =     DC_NCOND      $

.   =     uBRCMP   $      !if r1 = r0 fpc <- pc + lab
d_branchlength   =     DB_IMM;
d_fpcctl      =     DF_COND;
d_stall              =     DS_R0R1;
d_cond               =     DC_CMP   $

.   =     uBRNCMP $       !if r1 != r0 fpc <- pc + lab
d_branchlength   =     DB_IMM;
d_fpcctl      =     DF_COND;
d_stall              =     DS_R0R1;
d_cond               =     DC_NCMP $

.   =     uSBRCOND       $      !if cond(r1) fpc <- pc + lab,squash
d_fpcctl      =     DF_COND;
d_squash      =     DQ_COND;
d_stall              =     DS_R1;
d_cond               =     DC_COND $

.   =     uSBRNCOND    $      !if ncond(r1) fpc <- pc + lab,squash
d_fpcctl      =     DF_COND;
d_squash      =     DQ_COND;
d_stall              =     DS_R1;
d_cond               =     DC_NCOND      $

.   =     uSBRCMP  $      !if r1 = r0 fpc <- pc + lab,squash
d_branchlength   =     DB_IMM;
d_fpcctl      =     DF_COND;
d_squash      =     DQ_COND;
d_stall              =     DS_R0R1;
d_cond               =     DC_CMP   $

.   =     uSBRNCMP       $      !if r1 != r0 fpc <- pc + lab,squash
d_branchlength   =     DB_IMM;
d_fpcctl      =     DF_COND;
d_squash      =     DQ_COND;
d_stall              =     DS_R0R1;
d_cond               =     DC_NCMP $

.   =     uBRTAGEQ       $      !if tageq(r1,tag) fpc <- pc + lab
d_fpcctl      =     DF_COND;
d_stall              =     DS_R1;
d_cond               =     DC_TAGEQ       $

.   =     uBRTAGNEQ    $      !if tagneq(r1,tag) fpc <- pc + lab
d_fpcctl      =     DF_COND;
d_stall              =     DS_R1;
d_cond               =     DC_TAGNEQ     $

.   =     uSBRTAGEQ      $      !if tageq(r1,tag) fpc<- pc + lab,squash
```

```
          d_fpcctl    =    DF_COND;
          d_squash    =    DQ_COND;
          d_stall          =    DS_R1;
          d_cond           =    DC_TAGEQ      $

     .    =    uSBRTAGNEQ   $     !if tagneq(r1,tag) fpc<-pc + lab,squash
          d_fpcctl    =    DF_COND;
          d_squash    =    DQ_COND;
          d_stall          =    DS_R1;
          d_cond           =    DC_TAGNEQ     $

end
```