Monaco: A High Performance Implementation of FGHC on Shared-Memory Multiprocessors

S. Duvvuru

CIS-TR-92-16
July 1992

Abstract

The Monaco system is an attempt bridge the gap between state-of-the-art compilation and implementation of flat committed-choice logic programming languages like Flat Guarded Horn Clauses (FGHC). The Monaco system is unique in several respects. The abstract machine was designed in an attempt to steer away from traditional WAM-inspired abstract machine designs. The Monaco compiler utilizes newer compilation techniques such as decision graphs, as well as the more traditional optimizations such as dataflow analysis, to produce efficient code. We have completed a native-code implementation of Monaco on the Sequent Symmetry, a commercially available shared-memory multiprocessor. This report describes the design and evaluation of the Monaco system on the Symmetry, and compares it with other systems in the literature.

This report constitutes a thesis for the degree of Master of Science, submitted June 1992.

Department of Computer and Information Science
University of Oregon

1 Introduction

Within the past few years parallel computing has moved from the research laboratory into the computing centers and even into personal workstations. While new and efficient parallel architectures have continually emerged, the techniques used to program these new machines have not changed much from those used on sequential computers. Due to the lack of clean abstraction in traditional languages for specifying parallelism, the programmer is left with the task of harnessing the power of parallelism with little support from the programming language. Due to the lack of standards, programs are written using idiosyncratic thread libraries, making the programs not portable to a different or new architecture. Adding to the complexity of writing parallel programs, debugging them for correctness and performance is much harder than it is for sequential programs. Developing and debugging parallel programs will be greatly facilitated if the complexity of managing parallelism is kept hidden from the programmer. Sequential logic languages such as Prolog, which have clean declarative semantics, have been extended in several ways to fulfill the need for a good parallel programming language.

Removing the responsibility for managing parallelism only moves the responsibility on to the programming language. Parallel logic programming languages need optimizing compilers and efficient implementation to deliver high performance. Monaco is an effort at high performance implementation of Flat Guarded Horn Clauses (FGHC), a *committed-choice* logic programming language, on shared-memory multiprocessor systems.

Shared-memory multiprocessor machines are generally more efficient and easier to program than message-passing machines. However, with the current technology, message-passing architectures are be used to build scalable high performance computers. Hybrid architectures consisting of a network of shared-memory computers, present a good compromise between scalability and efficiency. Communication within a node is through shared-memory. Scalability is achieved by using an interconnection network. Building efficient systems on shared-memory is still a key to realizing the totally scalable parallelism. Our research contribution is in the efficient implementation of a parallel programming language on shared-memory multiprocessor architecture.

1.1 Committed-Choice Logic Programming

The success of Prolog as a sequential programming languages is due to the convenience of *unification* and *backtracking*. Full or *two-way* unification allows queries to be specified in both ways, i.e., find the solution for a problem; or, given a solution, find the inputs that describe the problem. Backtracking allows all the solutions to be found. Concur-

rent logic languages have sacrificed backtracking in favor of improved performance. The absence of backtracking suggests a name for this class of languages, *committed-choice* languages. The sacrifice of backtracking may be justified because most *real* problems require one solution; and if necessary, the search for all solutions can be simulated in programs that need to find all the solutions. Thus the expressive power of the language is not weakened.

A *committed-choice* logic program is a set of guarded Horn clauses of the form: "H :- $G_1, \ldots, G_m \mid B_1, \ldots, B_n$" where $m \geq 0$ and $n \geq 0$. H is the clause head, G_i is a guard goal, and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). When the guard is empty, the commit operator is omitted. "Flat" committed-choice languages have a further restriction that guard goals are simple builtin functions, such as $=, \leq, \geq, \neq$.

We say that a goal g commits to a clause i, if g successfully *matches* with the head of clause i (i.e., without causing any bindings to the variables of the goal) and the guards of clause i succeed without binding any goal variable. When a goal can commit to more than one clause in a procedure, it commits to one of them non-deterministically (the others candidates are thrown away). Structures appearing in the head and guard of a clause cause *suspension* of execution if the corresponding argument of the goal is not sufficiently instantiated. For example, in order for a goal `foo(X)` to commit to the clause "`foo(X) :- X = [A|B] | bar(A,B),`" the argument `X` of the goal must already be bound to a list structure, whose head (car) and tail (cdr) may be any term, even unbound variables.

A suspended invocation may be *resumed* later when the variable associated with the suspended invocation becomes sufficiently instantiated. A program successfully terminates when, starting from an initial user *query* (a conjunct of atoms), after some number of reduction steps, no goals remain to be executed, nor are suspended.

1.2 Literature Review

In this section we briefly discuss some implementations of committed-choice logic programming languages. The family of committed-choice languages is quite closely knit, with few practical differences in expressive power. However, some languages provide features which make particular programming tasks, such as systems programming or meta-programming, easier.

Panda[28] is an emulator-based parallel implementation of KL1 on shared-memory multiprocessors. KL1 [36] is an extension of FGHC, including some powerful meta-logical features and support for writing systems software. KL1-B is the abstract machine instruction set for specifying the execution of KL1 programs. Panda introduced

independent scheduling queues with depth-first scheduling, and on-demand load balancing schemes. An evaluation of the Panda implementation and design choices is presented by Sato[27].

JAM Parlog is a highly efficient emulator implementation of Jim's Abstract Machine for Parlog [7]. It includes support for Or-Parallel execution and *deep* guards. This introduces additional overheads for process management, since a process activation tree must be maintained for deep guards. JAM uses specialized data areas to store different types of data structures, in addition to the heap. For example, the *process stack* is used to store fixed size goal records, and the *argument stack* area is used to store process arguments and environments. All other dynamically created data structures are stored in the *heap* area. The JAM Parlog system implements private process queues which are also made accessible to other processors. Load balancing is achieved by allowing idle processors to steal work from busy processors. Different local scheduling schemes allow the private queue to function either in a LIFO or FIFO fashion. JAM Parlog also implemented a form of tail recursion optimization, by executing the "last" body goal directly, instead of creating a process for it. A *timeslicing* scheme is also introduced to prevent a single goal from monopolizing a processor, when a LIFO local queue management scheme is used.

Strand[11] is a distributed-memory implementation of Flat Parlog, with *assignment*. The assignment operator `:=/2` can be used by the programmer whenever it is known that one of the terms in an unification will be a variable. The simplifications in the supported language allow an efficient implementation. Other optimizations include simplified data structures to implement suspension, based on the observation that most goals suspend on only one variable at a time, and better compilation of process definitions.

Concurrent constraint languages are generalizations of concurrent logic programming, primarily in that unification becomes general constraint matching. One of the first CCL implementations is Janus, a committed-choice language [26]. Janus has the restriction that a stream can have one and only one producer and consumer. These channels allow implementation optimizations that are normally very difficult to do for standard committed-choice languages, such as local memory reuse. An experimental Janus compiler `jc` has been recently developed [14] based on decision-graph code generation, as is our Monaco compiler. Unlike Monaco, `jc` produces sequential C code, which is then compiled.

1.3 The Monaco Project

The Monaco project consists of: 1) design of an abstract machine; 2) construction of an optimizing compiler; 3) implementation of a multiprocessor run-time system; and 4) performance evaluation. The Monaco compiler development was effectively decoupled from all implementation detail, when the design of the abstract machine was finalized. A schematic diagram showing the flow of a FGHC source program through the various parts of the compiler is shown in Figure 1. Like the optimizing compilers for other languages, compilation is a multi-phase process with optimizations performed at each phase. It is beyond the scope of this thesis to describe the workings of the compiler. The Monaco compiler is described in a forthcoming paper [34].

The main contributions of this thesis are: We have successfully completed a native code implementation of a flat committed-choice logic programming language on a shared-memory multiprocessor. We have introduced the concept of *millicode* procedures to implement complex instructions in the run-time system. The viability of the instruction set for compiling flat committed-choice logic programs is demonstrated. The working of our optimizing compiler is experimentally verified. The preliminary performance benchmarking indicates that Monaco compares well with some of the well-known and efficient implementations.

This thesis presents the design of the Monaco abstract machine, implementation of the run-time system on Sequent Symmetry and performance evaluation. Section 2 describes our design of the Monaco abstract machine. Section 3 describes the abstract machine implementation on the Sequent Symmetry. Section 4 describes the design and implementation highlights of the Monaco run-time kernel. Section 5 contains the results of a performance analysis, and comparison with similar systems discussed in the literature. In Section 6, we look back on our effort to identify the lessons we had learned, the major accomplishments, and suggest directions for future work.

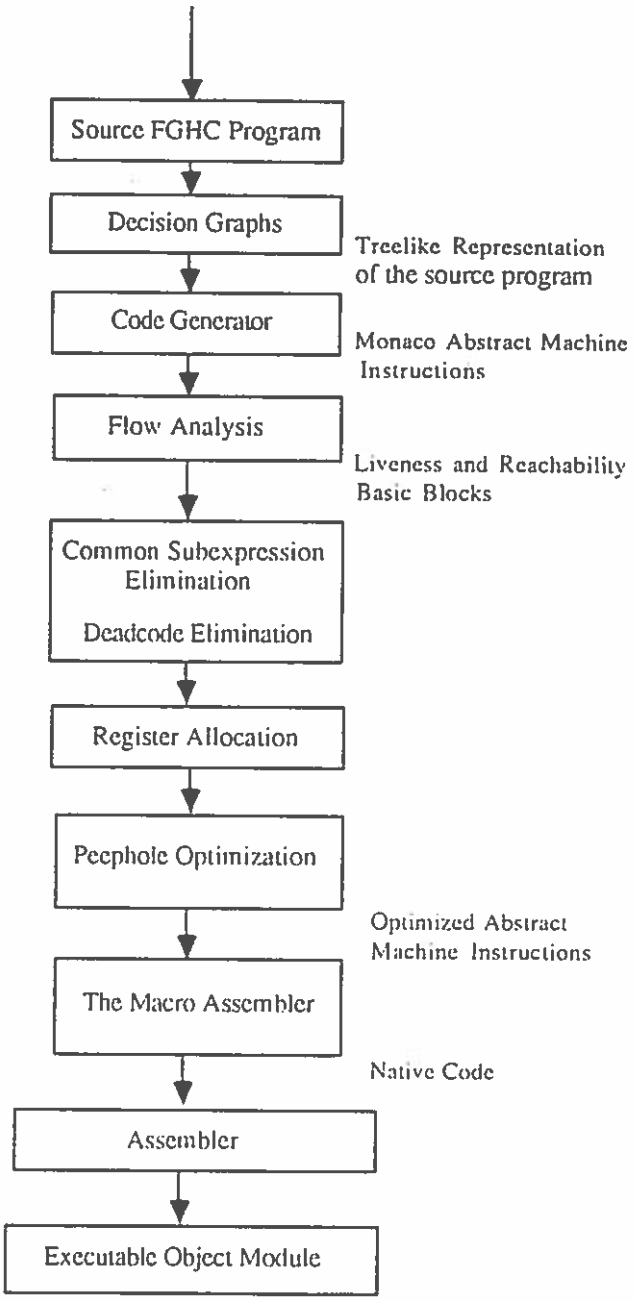

Figure 1: The Monaco Optimizing Compiler

2 The Monaco Abstract Machine

In this section we describe the Monaco abstract machine. Several abstract machines and instruction sets have been proposed for implementing parallel logic languages. Most of these are extensions of the basic Warren Abstract Machine [39] for process-based execution, e.g., JAM Parlog [7], Aurora [3], and Panda [28].

The Warren Abstract Machine (WAM) was designed to efficiently implement the backtracking and stack-oriented execution of sequential Prolog. One main contribution of WAM to Prolog implementation is the way unification is compiled. The emphasis on compiling unification is evident from the fact that a large percentage of WAM instructions are for specialized unification. Prolog supports two-way unification and specialization of unification is justified since Prolog programs spend considerable amount of time performing unification [9]. There are several successful implementations of the WAM in software using byte-code emulators, e.g., [25, 2]. Compiling Prolog programs to native-code has been attempted. Taylor presents results from a Prolog compiler for a RISC architecture in [31].

Committed-choice languages [29] differ from Prolog in many ways. Firstly, they have a process-based computation model that does not support backtracking. A computation consists of *reducing* goals, referred to as *processes*, until no unreduced goals remain, in which case the computation is said to *succeed*.

Secondly, unification is somewhat less complex, since only one-way unification is permitted in head matching and generalized two-way unification is permitted in the body. Also, there is a wide gap in efficiency of memory usage in sequential and parallel execution. In parallel execution. All memory is allocated on the heap and since there is no dynamic reclamation of memory as in sequential Prolog execution, the parallel programs consume large amounts of memory. Programs also pay a penalty in execution time for process management.

Since it's definition in 1983, the WAM has influenced the design of almost every abstract machine designed for logic languages. To our knowledge, there have been few successful efforts (e.g., Kursawe [22]) at exploring the design space without strong leanings towards the WAM. KL1-B, JAM, and Aurora are all examples of abstract machines designed by extending the WAM instruction set to support parallel execution.

At a very high level, all abstract machines are similar in the sense that they provide a clean abstraction of real program execution by making it easy to specify control flow and data manipulation. They may differ in the *power* of the abstract machine instructions or the ease with which the abstract machine may be mapped to a real machine. We have decided to try a fresh and simplified approach to abstract machine design. Our motivations for designing the Monaco abstract machine are summarized

below:

New compiling technologies

Compilation technology for logic programs has advanced since the design of the WAM. Although the WAM is clever at compiling unification, newer compilation techniques, such as decision graphs proposed by Kliger[20], greatly improve clause indexing in comparison to WAM. Debray's recent work [8] on weighted decision trees, global dataflow analyses such as Ueda's mode analysis [37] for committed-choice languages, Korsloot and Tick's thread sequentialization determinacy testing algorithms [21], and Zhong's granularity analysis algorithm [40] also have altered the design space for abstract machines. The productive interaction between compiler technology and instruction sets has been demonstrated by compilers for RISC architectures. The Monaco compiler incorporates some of the new compilation techniques and some traditional optimizations [1]. It is necessary to reevaluate and rethink many of the WAM-inspired strategies in the context of new compiler technology. This thesis presents an evaluation of such compiler technology.

New processors and architectures

The state of the art in processor design and computer architecture has advanced dramatically in the last ten years. We have witnessed a revolution brought on by the pipelined RISC processors. This trend has continued into superscalar and superpipelined processors [16]. Though not much has changed in the basic design of shared-memory multiprocessor architectures, we have seen that better cache protocols and advances in memory technology have removed some performance bottlenecks. Our goal was to design an instruction set that maps readily to a target machine for native-code implementations. We followed the RISC design approach: design a simple and orthogonal instruction set such that the most frequently executed instructions are implemented directly in the machine. Reducing the semantic content of abstract machine instructions leaves few heavy-weight instructions such as unification and process management.

Evaluation of the run-time system

Smart compilation of unification contributes to reducing the execution time in Prolog [9]. In addition to unification, process and memory management constitute a major chunk of the execution time for parallel implementations. Our goal is to measure the effects compiled unification, design of the run-time system, and native-code compilation on the performance of committed choice languages like

FGHC. Analysis of the time spent in various parts of the run-time system and in the user program, will help future redesign and optimize the heavily-used components. In this thesis we present preliminary measurements obtained from a first implementation.

Cleaner abstraction of the execution model

We want to design an instruction set that presents a clean abstraction of the execution model. The details of implementation and possible runtime optimizations are kept hidden from the programmer. This facilitates writing and debugging programs.

2.1 The Abstract Machine

An abstract machine is similar to a "real" computer architecture. An abstract machine definition consists of specifying an execution model, the visible machine state and registers, a set of supported data types, and an instruction set to operate on data. We treat the following topics separately in the following sections, by first describing the notion of a *variable* in committed choice logic programming.

2.2 Variables and Single Assignment Property

A variable in concurrent logic languages[1] has the *single-assignment* property, i.e., it can receive a *value* at most once. A *writer* is a process that tries to bind a value to a variable. A *reader* is a process that accesses a variable to read the stored value. The first writer to a variable can assign a value to the variable. The following writers to an instantiated variable succeed writing, only if the value they wish to write can be unified with the value already stored in the variable cell. Readers of the value of a variable will *suspend* on the variable, if it was not instantiated. When the variable does get instantiated, the writer must resume all the readers suspended on the variable. Shared variables together with the single assignment property provide elegant synchronization mechanisms. Foster [12], Shapiro [29], and Tick [33] present popular programming paradigms in concurrent logic languages.

2.3 The Execution Mechanism

A good description of the operational semantics of concurrent logic programs is given by Shapiro [29]. We present an informal description of computation in a flat concurrent

[1] More precisely, flat committed-choice languages with atomic variable-term unification and non-atomic term-term unification.

```
p(X) :- true     | X=[a|Y], p(Y).    (1)
q(X) :- X=[a|Y]  | q(Y)              (2)
q(X) :- X=[b|Y]  | q(Y)              (3)

?- p(X), q(X).
```

Figure 2: A Sample FGHC Program and Query

language such as FGHC. To illustrate the execution mechanism, consider the example in Figure 2. Assuming that the goals `p(X)` and `q(X)` were entered at the query, they are put in a *goal pool*. Each goal in the pool can be reduced in parallel. Goal reduction consists of matching the goal arguments with the head of a clause and then evaluating the guard tests. In the above example, `p(X)` may be reduced with clause (1) and `q(X)` may reduce with either clause (2) or (3). Performing head matching and executing the guard tests is called a clause *try*. When a clause try is successful, the clause *commits* and new processes for the body goals of the committed clause are added to the goal pool. If goal `q(X)` was tried with clause (2) or (3) before goal `p(X)` was executed, the guard test in either clause cannot proceed since `X` is still unbound and `q(X)` will suspend on `X`. When `p(X)` is executed it assigns the value `[a|Y]` to `X`, adds goal `p(Y)` to the goal pool and resumes `q` which suspended on `X`. Computation proceeds in this fashion until all the goals are reduced. If no goal can be reduced because all are suspended waiting for variables to be instantiated, it is is a *deadlocked* computation. A *failed* computation is one in which a process cannot be reduced with the head of any of the clauses in the procedure for the goal.

When head matching or a guard test cannot proceed due to an unbound variable, the variable is pushed onto a *suspension stack*. If at some point during the evaluation of a procedure, there are no more clauses left to try, the suspension stack is checked. If empty, it means that all guards have failed, hence the goal and consequently the computation should fail. If the suspension stack is *not* empty then the goal is suspended on all unbound variables found on the suspension stack.

2.4 The Data Types

The Monaco abstract machine supports seven programmer-visible tagged data types. The tag scheme of Monaco is similar to that employed in the original WAM design. The tag of a simple object (integers, symbols) is stored along with the object whereas the tag of a compound object (vectors, pairs) is stored in a pointer to the object rather than the object itself. This representation is efficient, since the type checking of an object may be done before the pointer to the object is dereferenced to access the object. For

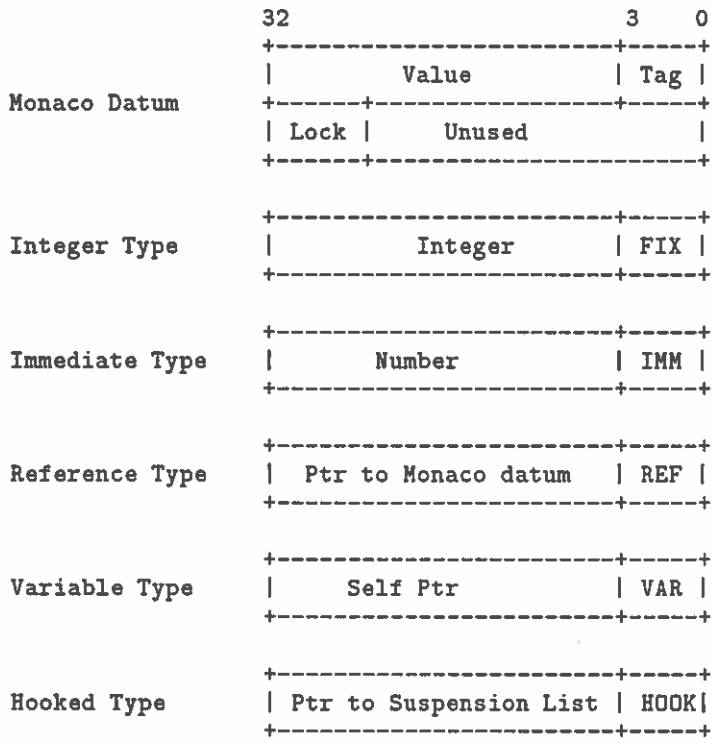

Figure 3: Simple Monaco Data Types

further discussion of tag schemes and run-time type checking see Steenkiste [30].

We suggest a layout of Monaco datum (illustrated in the top of Figure 3), as it is not expected to be different from one host machine to another. The suggested layout should suffice for any processor that supports 32-bit data and atomic access to a byte or word which is word-aligned. Each tagged datum occupies two 32-bit words. The first word contains the tag and value of the datum; the tag occupies the least significant three bits of the word and the rest of the word is used to store the value. Each datum is potentially shared and in a parallel implementation, races to instantiate a shared variable are possible. To remove unpredictable races, each cell is protected with a *lock*. The most significant byte of the second word is used to store the lock. Conceptually a bit is sufficient to implement a lock, but general-purpose processors do not support atomic bit access. The least significant three bytes in the second word are currently unused. Many of the issues in layout design are inter-dependent and we suggest that the following description be first read in its entirety.

Due to the number of supported data types, three bits are used to encode the tag. The tag is stored in the three *least* significant bits of of the first word. This is true even for the pointer data type `#REF`. (See the definitions of Monaco data types later in this

section). This restricts all Monaco data to be stored on an 8-byte boundary. Note that the Monaco datum, excluding the lock, will fit into one 32-bit word or four bytes. The 8-byte alignment would cause the second word to be wasted, if we did not use it for storing the lock. Storing the lock along with the datum is expected to improve cache performance because the lock variable for a datum is fetched into cache along with the datum. If the lock were not stored together with the datum, a datum would occupy one full word leading to more economical usage of the cache, and the locks could be stored in a table of locks. Since we need to quickly index into the table, a reference to the table must always be kept in a register; which cannot always be guaranteed. With our scheme, when a datum is fetched into the cache, the lock is also fetched and subsequent lock access is inexpensive. The alternative scheme may cause a cache miss while accessing the lock.

Three bits are needed to store the tag. The tag storage in the least significant bits of the datum makes the tag stripping operation inexpensive. We suggest that the tag code of `000` be assigned to `#REF` tag. No cost is involved in creating the `#REF` type datum, and since this type is used extensively, the tag scheme should work efficiently. In the following pages, we describe how each data type is laid out in memory. The second word in the datum which contains the lock, is not shown in the Figure 3.

`Integer`

An `integer (#FIX)` type datum holds a fixed-precision integer quantity. Equal integer values always have identical representations. The integer is stored in the 2's complement form.

`Immediate`

An `immediate (#IMM)` type datum represents an atom (a textual symbol); or miscellaneous constants such as the nil constant `[]` and vector and goal record headers, etc.

`Variable`

A `variable (#VAR)` cell is an uninstantiated location. When a variable becomes instantiated, its type changes to that of the value that the cell is instantiated with. The value part of a variable cell is set to point to itself.

`Hooked`

A `hooked cell (#HOOK)` cell is an uninstantiated location on which some processes are blocked. When a hooked cell is instantiated during unification, the processes hooked on the cell will be unblocked. The value part of the hooked cell points to a list of processes suspended on the cell.

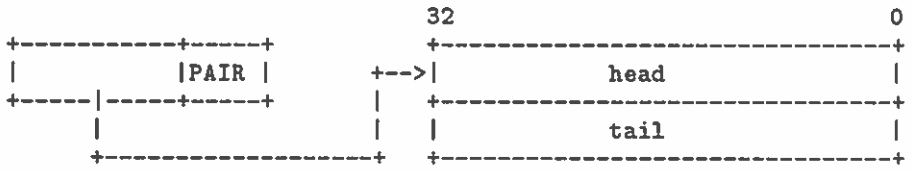

Figure 4: The Pair Data Type

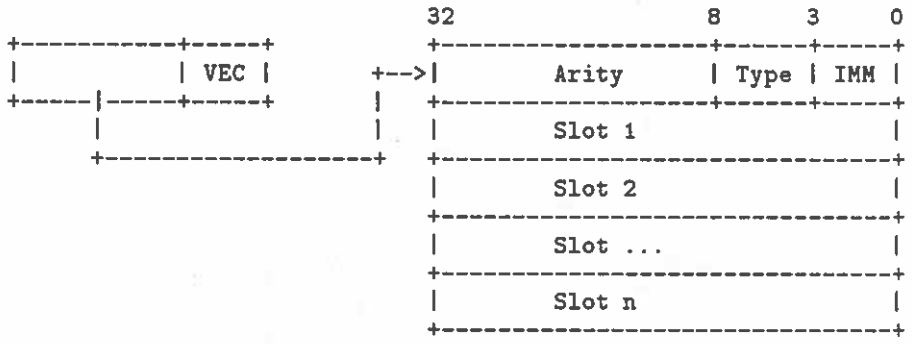

Figure 5: The Vector Data Type

`Reference`

A `reference (#REF)` contains a pointer to a tagged word.

`Pair`

A datum of type `pair (#PAIR)` points to a list cell (see Figure 4. List cells are represented by two consecutive data items. A list pointer points to the first item, denoted as the "head" of the list cell. The second item is denoted as the "tail" of the list cell. The terms "car" and "cdr" are sometimes used, to refer to the head and tail of a list cell, respectively. The list cell has no header.

`Vector`

A `vector (#VEC)` type datum holds a pointer to a vector of arity one or more (see Figure 5). A vector of arity n consists of space for n consecutive data items numbered 1–n. Slot zero of the structure contains the vector header (an immediate) in which n, the arity of the vector is encoded. The `Type` field distinguishes the particular use of a vector. Vectors can be used to represent both general structures and goal records.

2.5 The Machine Organization

The Monaco abstract machine is a general-purpose register-based architecture in which the only memory operations are loads and stores. We leave the size of the register set unspecified as it does not restrict the power of the abstract machine. A reasonable number of registers would be 16–32. Although we could have a larger register set, with a subset of them mapped to the target processor's registers, the strategy may not be efficient because all the processor registers would have to be saved on a context switch i.e., call to any of the run-time kernel routines such as **suspend**, at a significant cost.

2.6 The Instruction Set

While other low-level instruction sets for Prolog and it's derivatives exist (e.g., the BAM [17, 38] and CARMEL-2 [15]) they have been designed for hardware implementation. Our interest is in an instruction set which readily maps onto a conventional microprocessor architecture. We use our abstract machine as the target architecture for our compiler, and provide a simple translator from the abstract machine language to the host processor's assembly language.

Our goal was to obtain an orthogonal and low-level instruction set to serve was the target language for our optimizing compiler. The instruction set can be divided into the following categories: data constructors, data manipulators, predicates, arithmetic, control transfer, process management, and unification.

In terms of the *weight* or *semantic content*, the instructions may be categorized into three main categories. A majority of the instructions are lightweight and can easily be translated into small sequences of instructions on a host machine. Most predicate and arithmetic instructions fall into this category. At the other extreme, some instructions are sufficiently complex that not much may be gained by translating them into native code. Unification and process-management instructions are examples of this variety. The middle ground is covered by the data manipulators and constructors. Whether to implement this type of instructions as native code or procedure calls is left as an implementation detail. Each group of instructions is described separately below. All instructions are summarized in Appendix A.

In the following discussion, `Rs`, `Rs1`, and `Rs2` denote source registers, while `Rd` denotes a destination register. `n` denotes an integer constant. `<const>` denotes any constant in the source language, *i.e.*, immediates and fixed point numbers. `Size` is used to specify the size of a vector frame. `Proc/Arity` is a pointer to an executable procedure. The word *millicode* will be used to describe functions or instructions that are implemented in the run-time kernel, the user program will have to perform a kernel call in order to use the millicode procedure.

mkconst (<const>, Rd)	Rd := <const>
mklist (Rs1, Rs2, Rd)	Rd := #PAIR ptr to new list cell
mkstruct (Size, Rd)	Rd := #VEC ptr to new struct
mkgoal (Size, Proc, Rd)	Rd := #VEC ptr to new goal record for Proc
mkvar (Rd)	Rd := #REF ptr to new #VAR cell
alloc (n,Rd)	Rd := #REF ptr to n heap cells
ref (Rs, n, Rd)	Rd := #REF ptr to $(Rs + n)^{th}$ heap cell
initlist (Rs1, Rs2, Rd)	Rd := #PAIR ptr to list cell
initstruct (Size, Rd)	Rd := #VEC ptr to a struct
initgoal (Size, Proc, Rd)	Rd := #VEC ptr to goal record for Proc
initvar (Rd)	Rd := #REF ptr to a #VAR cell

Table 1: Data Constructors

2.7 Data Constructors

Data constructors provide a means to create data structures and values in an implementation independent fashion. These instructions are listed in Table 1. We do not provide constructors for all data types defined in the abstract machine. The `mklist`, `mkstruct`, `mkgoal` and `mkvar` instructions construct a new list cell, a new structure frame, and a new goal record, on the heap, respectively. The `mk` class of instructions are sufficient for constructing all the data types.

Each of the `mk*` instructions invoke the heap memory allocator. If heap data structures are created frequently in a program, the overhead of calling the heap allocator can be considerable. It may be economical to reserve the heap space required by a program segment in a single call, if possible. We extended the set of data constructors with the following instructions, in order to encourage such a style of compilation.

The `alloc(n,Rd)` instruction reserves `n` cells on the heap. This instruction is intended to reduce the number of calls to the memory allocator. The number of cells required by a program segment can be determined at compile time and allocated in a single call. This is similar to the notion of creating a call frame for the local variables of a called function in procedural language implementations. The `ref(Rs, n, Rd)` instruction is used to index into a block of heap memory reserved in an `alloc` instruction.

The `initlist`, `initstruct`, `initgoal` and `initvar` instructions construct data structures in heap space which is already allocated, to which a reference pointer is available in `Rd`. The `init` class of instructions not only avoid having to allocate new

deref (Rs, Rd)	Rd := deref(Rs)
move (Rs, Rd)	Rd := Rs
sset (Rs1, n, Rs2)	Slot n of Rs2 := Rs1
sref (Rs, n, Rd)	Rd := Slot n of Rs
ssize (Rs, Rd)	Rd := Size of struct in Rd
car (Rs, Rd)	Rd := car(Rs)
cdr (Rs, Rd)	Rd := cdr(Rs)
setcar (Rs, Rd)	car(Rd) := Rs
setcdr (Rs, Rd)	car(Rd) := Rs

Table 2: Data Manipulators

memory and checking for heap overflow, but also will drive more sophisticated structure reuse schemes. See the discussion on memory reuse in Section 6.

2.8 Data Manipulators

Data manipulators access and update data structures. In general, the type of the data structure being updated is encoded in the pointer to the structure, and hence type checking can be done. However, range checking is not performed in general. See Table 2

The `setcar` and `setcdr` instructions are currently not emitted by the compiler. They are intended to facilitate reuse of list cells when the structure reuse analysis phase in the compiler is implemented.

2.9 Predicates

A predicate leaves a truth value in its destination register; the truth value is a tagged integer, with `FIX#1` representing true and `FIX#0` representing false. Type predicates and general predicates are shown in Table 3; type predicates are used to check the tags of tagged words. Integer predicates are shown in Table 4; if an integer predicate is passed a non-integer, it raises an exception.

2.10 Integer Arithmetic and Bit Operations

Integer operations take tagged integers as arguments and produce tagged integers as results. Non-integer operands will cause an exception. In the case of arithmetic operations, an overflow will cause an exception. Integer operations are shown in Table 5.

`isnil (Rs, Rd)`	`Rs == [] ? Rd = True : Rd := False`
`isint (Rs, Rd)`	`tagof(Rs) == #FIX ? Rd := True : Rd := False`
`isref (Rs, Rd)`	`tagof(Rs) == #REF ? Rd := True : Rd := False`
`isunbound (Rs, Rd)`	`tagof(Rs) == #VAR ? Rd := True : Rd := False`
`ishooked (Rs, Rd)`	`tagof(Rs) == #HOOK ? Rd := True : Rd := False`
`islist (Rs, Rd)`	`tagof(Rs) == #PAIR ? Rd := True : Rd := False`
`isstruct (Rs, Rd)`	`tagof(Rs) == #VEC ? Rd := True : Rd := False`
`isimm (Rs, Rd )`	`tagof(Rs) == #IMM ? Rd := True : Rd := False`
`eq (Rs1, Rs2, Rd)`	`Rs1 == Rd2 ? Rd := True : Rd := False`
`neq (Rs1, Rs2, Rd)`	`Rs1 == Rd2 ? Rd := False : Rd := True`

Table 3: Type Predicates and General Predicates

`ieq (Rs1, Rs2, Rd)`	`Rs1 == Rs2 ? Rd := True : Rd := False`
`ineq (Rs1, Rs2, Rd)`	`Rs1 == Rs2 ? Rd := False : Rd := True`
`ilt (Rs1, Rs2, Rd)`	`Rs1 < Rs2 ? Rd = True : Rd = False`
`ile (Rs1, Rs2, Rd)`	`Rs1 <= Rs2 ? Rd = True : Rd = False`
`igt (Rs1, Rs2, Rd)`	`Rs1 > Rs2 ? Rd = True : Rd = False`
`ige (Rs1, Rs2, Rd)`	`Rs1 >= Rs2 ? Rd = True : Rd = False`

Table 4: Integer Predicates

2.11 Control Transfer Instructions

For the control transfer instructions, the destination of transfer is specified by a symbolic identifier `Label` to denote a program point. See Table 6.

2.12 Unification Instructions

Two types of unification operations are required: *ActiveUnify* to perform two-way unification of arbitrary structures; and *PassiveUnify* to perform one-way unification in a clause try. Active and passive unification are complex operations and will nearly always be implemented as millicode calls. See Table 7 for the format of unification instructions.

Instruction `punify(Rs1,Rs2,Rd)` will return the result of passive unification as a

iadd (Rs1, Rs2, Rd)	Rd = Rs1 + Rs2
isub (Rs1, Rs2, Rd)	Rd = Rs1 - Rs2
imul (Rs1, Rs2, Rd)	Rd = Rs1 * Rs2
idiv (Rs1, Rs2, Rd)	Rd = Rs1 / Rs2
imod (Rs1, Rs2, Rd)	Rd = Rs1 % Rs2
ineg (Rs, Rd)	Rd = - Rs1
iand (Rs1, Rs2, Rd)	Rd = Rs1 AND Rs2
ior (Rs1, Rs2, Rd)	Rd = Rs1 OR Rs2
ixor (Rs1, Rs2, Rd)	Rd = Rs1 XOR Rs2
inot (Rs, Rd)	Rd = NOT Rs

Table 5: Integer Arithmetic and Bit Operations

jump (Label)	unconditional branch to Label
br (z, Rs, Label)	branch to Label if Rs == 0
br (nz, Rs, Label)	branch to Label if Rs != 0
br (p, Rs, Label)	branch to Label if Rs > 0
br (n, Rs, Label)	branch to Label if Rs < 0

Table 6: Control Transfer

unify (Rs1, Rs2)	active unify Rs1 with Rs2
punify (Rs1, Rs2, Rd)	passive unify Rs1 with Rs2

Table 7: Unification Instructions

tagged integer code in Rd. The result PUNIFY_SUCCESS indicates that the unification was successful. PUNIFY_FAILURE indicates that the arguments could not be passively unified. PUNIFY_SUSPEND indicates that the result of unification could not be determined since some variable(s) were not sufficiently instantiated. The unbound variables are pushed onto the suspension stack.

The *ActiveUnify* instruction aunify(Rs1,Rs2) does not have an explicit return value. If it fails then the entire computation is in error, an exception is raised, and the system is halted. Whenever aunify returns to the caller, it indicates that the active unification was successful.

We have avoided introducing customized unification instructions for the following reasons. Most calls to unify are to assign a value to a freshly allocated variable. If the compiler knows the data type of objects being unified, a specialized unify instruction can be more efficient than a call to general unification. But usually the compiler *can* know more: it can be derived statically that simple assignment suffices for an unification since one of the objects is guaranteed to be unbound. If the variable was created within the body of the goal, no other process has a reference to the variable, and it can be simply assigned to. The interesting situations arise when the variable is *not* created within the body of the clause. In *properly moded* FGHC programs [37], which constitute a majority of legal FGHC programs, there is a single writer to a variable. Mode analysis [23] can be done at compile time to identify the writer. The assign(Rs1,Rs2) will be generated instead of an unify instruction in such a case. The assign(Rs1, Rs2) instruction dereferences Rs1, and writes Rs2 into the dereferenced location. Synchronization and some tag checking are required to perform this operation correctly, in case some process is concurrently suspending on the same variable. However, since the location is guaranteed to be unbound, the complexity of the assign is less than that of the unify(Rs1, Rs2) instruction. The assign instruction will be used wherever possible, defaulting to general unify in the remaining cases.

The punify instruction performs passive unification and is only emitted when two head arguments of a clause are the same. Few clauses in *real* programs use shared head variables.

2.13 Process Management

The enqueue(Rs) instruction adds a goal, a pointer to which exists in Rs, to the ready-goal pool. The proceed instruction is used to signal the run-time system that a process has terminated. The suspend(Proc, N) instruction suspends a procedure Proc of arity N on all unbound variables placed on suspension stack. The push(Rs) instruction is used to place reference-type pointers to an unbound variable on the suspension stack.

enqueue (Rs)	enqueue goal pointed to by Rd
proceed	terminate calling process
suspend (Proc, N)	suspend Proc of arity N
execute (Proc, N)	call Proc/N
push (Rs)	put Rs on suspension stack

Table 8: Process Management Instructions

See Table 8 for a list of process management instructions.

2.14 A Quick Example: Quicksort

Quicksort is a version of the standard quicksort algorithm in FGHC. It sorts a list of integers, by partitioning the list of integers about a pivot element into two lists S and L containing numbers smaller and larger than the pivot element. The lists S and L are sorted independently and their sorted outputs are concatenated to form the completely sorted list. We do not present the partition procedure here. The Quicksort program and the code generated for it, by the Monaco compiler, is shown in Figure 6.

Before entry to a procedure Proc/N, the N arguments are loaded in to the abstract machine registers numbered 0 through N-1. The entry point to the compiled procedure is labeled qsort/3. All arguments are dereferenced before they are tested. Quicksort requires that the first argument is bound. If it is not bound, the first argument is pushed onto the suspension stack and the process suspends. If the first argument is bound to [], then the first clause commits and Rest is unified with Ans. The goal returns to the run-time system by executing proceed to indicate successful reduction.

If the first argument is not [], it is expected to be a list. The tag of the dereferenced first argument is then checked to see that it has a #PAIR tag. If it did not have this tag, the suspend instruction is executed with no variables placed on the suspension stack, which causes failure.

If the first argument has a #PAIR tag, the second clause commits. The alloc instruction is used to allocate a 15-cell frame on the heap, since that many cells are required in the ensuing code. Two goal records are constructed for the two qsort/3 goals in the body, and are enqueued. A call is made to the partition/4 procedure, after setting up its arguments in appropriate registers.

The call to one of the body goals in a clause is done with the execute instruction. This avoids the construction of a goal record and the subsequent overhead of process management for that goal. In case of Quicksort, the last call is made to partition/4,

```
qsort([],Rest,Ans) :- true | Rest = Ans.
qsort([X|R],Y,T)   :- true | partition(R,X,S,L),
                             qsort(S,Y,[X|Y1]),
                             qsort(L,Y1,T).

qsort/3:
          deref    r0 r4          % r4 := deref(Argument 1)
          isbound  r4 r3          % Bound(Argument 1)
          br       nz r3 L1003    % Yes.
          push     r0             % No. Push Argument 1. Suspend.
L1010:
          suspend  qsort/3        % Suspend
L1003:
          isnil    r4 r0          % Argument 1 is [] ?
          br       z r0 L1005     % No. Go to Clause 2.
          unify    r1 r2          % Yes. Rest = Ans
          proceed
L1005:
          islist   r4 r0          % Type(Argument 1) is #PAIR ?
          br       z r0 L1010     % No. Failure!
          alloc    15 r0          % Allocate 15 cells on heap.
          move     r0 r6          %
          initgoal 4 qsort/3 r6   % r6 -> goal qsort/3
          ref      r0 6 r5        % r5 := L
          initvar  r5
          sset     r5 1 r6        % 1(r6) := L
          ref      r0 7 r3        %    r3 := Y1
          initvar  r3
          sset     r3 2 r6        % 2(r6) := Y1
          sset     r2 3 r6        % 3(r6) := T
          enqueue  r6             % enqueue qsort(L,Y1,T)
          ref      r0 8 r2
          initgoal 4 qsort/3 r2   % r2 -> goal qsort/3
          ref      r0 14 r7       % r7 := S
          initvar  r7
          sset     r7 1 r2        % 1(r2) := S
          sset     r1 2 r2        % 2(r2) := Y
          car      r4 r6          %    r6 := X
          ref      r0 15 r8
          initlist r6 r3 r8       %    r8 := [X|Y1]
          sset     r8 3 r2        % 3(r2) := [X|Y1]
          enqueue  r2             % enqueue qsort(S,Y,[X|Y1])
          cdr      r4 r8          % r8 := R
          move     r8 r0          % r0 := R
          move     r6 r1          % r1 := X
          move     r7 r2          % r2 := S
          move     r5 r3          % r3 := L
          execute partition/4     % call paritition(R,X,S,L)
```

Figure 6: Compiled Quicksort Procedure in Monaco

which is the producer of data structures needed by the two `qsort/3` goals. This optimization on the last call, sometimes known as "tail recursion optimization," allows control to remain in the user program without frequently returning to the run-time kernel. If the compiler can analyze the producer-consumer relationships, and choose the right body goal for last call, this optimization can be used to keep the producers running ahead of consumers, thereby reducing the frequency of suspension. The Monaco compiler presently chooses the first body goal, occurring in textual order. Note that if none of the called procedures suspend, the last-call optimization could lead to a long thread of procedure invocations, without returning to the run-time system. We show how this can be harmful, and how it is avoided, in the next section.

Programmable Registers
%eax %ebx %ecx %edx
%esi %edi
%esp %ebp

Table 9: The intel 386 Register Set

3 The Realization of Monaco Abstract Machine

In this section we describe how the Monaco abstract machine is realized on the intel 386 processor and implemented on the Sequent Symmetry shared-memory multiprocessor. We will briefly describe the intel 386 processor, the Sequent Symmetry, and how the abstract machine is mapped onto the host machine. We also describe an optimizing macro assembler that translates abstract machine instructions into short sequences of intel 386 assembly language instructions.

3.1 The intel 386

The intel 386 processor is an advanced CISC microprocessor designed for multi-tasking. It supports both 32-bit data and addresses, a segmented memory model and virtual memory management support. The complexity of memory and protection schemes is hidden from user programs by not allowing them to operate on the segment and control registers. See Table 9 for a list of programmable registers. Note that several intel 386 instructions have implicit operands for e.g., `%esp` is an implicit operand to the `push` instruction and is expected to always point to the top of the program stack. This rules out the general availability of the `%esp` register for holding any other information. The `mul` and `div` instructions are also examples of instructions with implicit operands. See the section on Application Programming in [18] for a detailed description of the intel 386 instruction set.

Of the eight programmable registers, we decided not to use `%esp` and `%ebp` since they are used by the C procedure calling mechanisms. This leaves only six registers which are available to Monaco. The available registers are so few that most compilers are not effective in keeping frequently-used variables in registers. The reader who is familiar with large register files available on RISC architectures would find this situation rather suffocating! Obviously, the few available registers must be used with economy. One of the important steps in an implementation is to decide how the registers will be used to realize the Monaco abstract machine.

GLBL	%esi	pointer to globals array
SSTK	%edi	pointer to suspension stack
REGS	%edi	pointer to rootable register set
TMP0	%eax	scratch register 1
TMP1	%ecx	scratch register 2
TMP2	%ebx	scratch register 3

Table 10: The Register Map

3.2 The Sequent Symmetry

The Sequent Symmetry is a shared-memory multiprocessor system. The S81 configuration consists of up to 20 intel 386 processors operating on a *shared* bus. All the memory is shared and accessible to all processors *via* the shared bus. A possible bottleneck to performance and scalability is the restricted bandwidth of the shared bus. Reducing bus traffic is a key to achieving good performance. A 128 KB instruction and data cache is used to exploit memory reference locality thus reducing the bus traffic. The cache coherency protocol employed is *copyback* with a *write invalidate* policy. Multiple *read-only* copies of a memory location are allowed to exist in different processor caches; the writing processor becomes the owner of the cache block to which the address belongs. A snoopy cache coherence protocol is used to invalidate all other cached copies of the memory location.

Sequent Symmetry runs the Dynix 3.1 operating system which is derived from the 4.2 BSD version of Unix. The operating system support for microtasking is sufficient for large-grain parallelism; but for the fine-grained parallelism prevalent in a process-based parallel logic programming language, it is more sensible to manage the parallelism without the process-creation system calls of Dynix, such as `fork` and `mfork` [24].

3.3 Mapping the Abstract Machine

Implementing an abstract machine is directed mainly by the available resources on the host machine. Apportioning the physical resources in such a way as to obtain a clean and efficient implementation is a process of stepwise refinement. In the following sections we describe the data structures required by the abstract machine and to some of which, quick access is needed. These data structures or references to them must be loaded in to the registers of the host machine before a process is executed. The tentative register mapping is given in Table 10.

This particular register mapping was obtained after noting that the most frequently-used data structure is the abstract machine register set, in which all computations take place. The heap top pointer and heap limit are held in the globals array. Due to the heavy memory allocation, the globals array is frequently accessed. Also used heavily, though not to the same extent as the earlier two, are the *per* processor suspension stacks. Three registers REGS, GLBL, SSTK are set up with the base addresses of the register set, globals array, and the suspension stack, respectively. Three registers are reserved to hold temporary results when the abstract machine instructions are macro expanded into sequences of host machine instructions. This mapping appears to be intuitive, it should be clear that it is not the only possible register map. Perhaps more detailed performance studies in future will lead to a better register mapping.

3.4 The Register Set

The Monaco abstract machine operates with some fixed number of registers. These are stored in the `registers` array for each processor. Arguments to a procedure are loaded into these registers before the procedure is entered. A pointer to the base of this array is kept in the physical register REGS for efficient access.

3.5 The Millicode Table

Some of the abstract machine instructions are sufficiently complex to require implementation in millicode (e.g., `suspend`, `enqueue`). Since the addresses of millicode procedures are not known at compile time, a millicode call is accomplished by means of an indirect `jmp` instruction. A table of entry addresses for all millicode procedures is needed for dynamic linking. The run-time system initializes the *millicode table* with the addresses of all millicode procedures. A millicode call is accomplished by pushing the return address and parameters in the usual stack-parameter passing style of C, and then executing an indirect `jmp` to the start address of the millicode. The millicode table is appended at the end of the `registers` array, because that makes the millicode table accessible via the REGS physical register. The millicode call mechanism is illustrated with the example of `punify` in Figure 9, described later in this section.

3.6 The Suspension Stack

The suspension stack is a data structure where unbound variables that potentially suspend execution are placed. The abstract machine instruction `push` places pointers to uninstantiated variables on the suspension stack. A suspension stack is allocated for each worker. A pointer to the suspension stack structure is set up in the SSTK physical

```
# template for move(Rs,Rd)
movl    Rs*4(REGS),TMP0
movl    TMP0,Rd*4(REGS)
```

Figure 7: Macro Expansion of `move(Rs, Rd)`

register.

3.7 The Timer

The compiler performs "last call optimization," i.e., the call to the first procedure is converted to an `execute(Name/Arity)` instruction. This instruction jumps directly to the target procedure assuming that the registers are set up correctly with the arguments in preparation for the last call. This optimization saves the overhead of creating and queuing a goal record for one of the body goals.

The savings can be substantial since the kernel is entered fewer times. However, this could lead to infinite loops in uniprocessor execution and sometimes wasteful computation in multiprocessor execution; with producers running ahead of consumers and using up memory. In order to reduce wasteful computation, a countdown timer is initialized with a suitable constant; it is decremented and checked at each `execute` instruction to see if it has reduced to zero. If this counter ever becomes zero, then the `execute` instruction will make a goal record for the procedure, enqueue the goal, and return to the kernel. This will prevent any processor from being monopolized by a single process. The `execute` instruction will be as expensive as an `enqueue` instruction in the event of the timer reducing to zero. This is fairly rare, dependent on the initial value of the countdown timer.

The timer decrement is performed once at the end of a procedure, and is relatively inexpensive. Due to the shortage of registers, we do not dedicate a register to hold the timer. The Timer is appended to the Millicode table.

3.8 Macro Expansions

The assembler program on Unix, *as*, is used to assemble the intel 386 code into executable machine code. The expansion of the abstract machine instructions into assembly language instructions is simple. Due to lack of space, we will only present a few macro expansions to give a feel for the expansion style. All constants and expressions are evaluated by the assembler.

The macro expansion for `move(Rs, Rd)` instruction is shown in Figure 7. The `move` instruction needs two assembly instructions, since memory-memory moves are not

```
        # template for deref(Rs,Rd)

        movl    Rs*4(REGS),TMP0
        movl    TMP0,TMP2       #  TrailPtr := t
L(k):
        movl    TMP0,TMP1
        andl    $TAG_MASK,TMP1  # while(TAG(t) == REF_TAG)
        jnz     L(k+1)          # {
        movl    TMP0,TMP2       #    TrailPtr = t;
        movl    0(TMP0),TMP0    #    t = CELLREF(t);
        jmp     L(k)            # }
L(k+1):
        movl    TMP0,Rd*4(REGS) # Rd = Result
```

Figure 8: Macro Expansion for `deref(Rs,Rd)`

```
        # template for punify(Rs1,Rs2,Rd)

        pushl   Rs2*4(REGS)         # push Rs2 on stack
        pushl   Rs1*4(REGS)         # push Rs1 on stack
        pushl   $LM_RETURN          # push return address
        jmp     *PUNIFY_OFF(REGS)   # indirect call to punify

LM_RETURN:
        addl    $8,%esp             # readjust stack
        movl    TMP0,Rd*4(REGS)     # Rd := punify(Rs1,Rs2)
```

Figure 9: Macro Expansion for Millicode call to `punify`

permitted. Often register `Rs` is already in `TMP0` as the result of a previous operation, and the first `move` from registers kept in memory into `TMP0` may be avoided by a *peephole* optimizer.

The macro expansion for the `deref(Rs,Rd)` instruction is shown in Figure 8. The `deref` instruction is one of the most complex expanded instructions. The accompanying comments give the pseudo-code for the dereference operation. For clarity we do not show how the `#HOOK` type datum, which points to a suspended goal, is handled.

The macro expansion for `punify(Rs1,Rs2,Rd)` instruction is shown in Figure 9. The millicode call method is illustrated here. The source operands `Rs1`, `Rs2`, and the return address are pushed on the C stack. The result is returned in `TMP0`, which is then stored in register `Rd`.

Other schemes where arguments and/or return address are placed in registers will be efficient, but there simply are not many registers to go around. The stack-based millicode call style will be used until a better method is devised.

```
# Naive expansion of mkconst      # Smart mkconst
                                  # N := 10*8 + FIX_TAG
movl    $10,TMP0
shll    $3,TMP0                   movl    $N,Rd*4(REGS)
orl     $FIX_TAG,TMP0
movl    TMP0,Rd*4(REGS)
```

Figure 10: Macro Expansion for `mkconst(10,Rd)`

```
 # car(R4,R1)

 movl   4*4(REGS),TMP0
 movl   -PAIR_TAG(TMP0),TMP0
 movl   TMP0,1*4(REGS)

 # move (R1,R3)
** movl   1*4(REGS),TMP0      # Not emitted!
   movl   TMP0,3*4(REGS)
```

Figure 11: The Memory Fetch Optimization

3.9 The Macro Assembler

The macro assembler is the penultimate stage in the compilation process. It takes as input the `*.mon` file produced by the compiler and produces a `*.s` for input to the Unix assembler program *as*. The macro assembler is also used to perform a final level of peephole optimizations at the assembler instruction level. Much of the translation is table driven and can be easily retargeted for a different host processor. We describe the important peephole optimizations in the macro assembler.

Tagged constants

The compiler often emits instructions like `mkconst(foo,Rd)` or `mkconst(10,Rd)`. A tagged representation of the constant is to be placed in register `Rd`. Figure 10 shows the naive and smart ways in which a tagged integer constant 10 is produced. In the smart version, the constant `N` is evaluated by the assembler itself and three instructions are saved.

The Memory Fetch optimization

Often the source register operand of an abstract instruction is already in one of the scratch registers of the host machine, and need not be fetched again from the abstract machine's registers which may be in main memory. Consider the Figure 11 in which a `car` instruction is followed by a `move` instruction. The contents of the `R1` register are already available in `TMP0` and hence `R1` need not be fetched again from memory.

```
        Unoptimized                          Optimized

        ieq     R1, R2, R3                   br   neq, R1, R2, L1000
        br      z,  R3, L1000


        #ieq     R1, R2, R3                  #br   neq, R1, R2, L1000
        movl     $FIX_TRUE,3*4(REGS)         movl  1*4(REGS),TMP0
        movl     1*4(REGS),TMP0              cmpl  TMP0, 2*4(REGS)
        cmpl     TMP0,2*4(REGS)              jne   L1000
        je       L100
        movl     $FIX_FALSE,3*4(REGS)

L100:
        #br      z,  R3, L1000
        cmpl     $FIX_FALSE,3*4(REGS)
        je       L1000
```

Figure 12: Unoptimized and Optimized Branching

Branch optimization or boolean evaluation for control

Boolean constants are given #FIX number representations in Monaco. See the description of predicate instructions in Section 2. Boolean predicates set the contents of the destination with the result of the test. This result is used by a following conditional branch instruction to transfer control. If the boolean constants are evaluated for setting a true or false condition code, and the result of the test is not used for effect, it is safe to not store the result of the predicate in the destination register.

Based on this idea, we allow the macro assembler to merge a predicate test and the following conditional branch into one single instruction as shown in Figure 12. The generated 386 code for the unoptimized and optimized versions is also shown. Note that the optimized code has five less instructions and makes fewer memory references. We are considering a possible improvement on this scheme by saving condition-code register of the 386 processor, whenever the result of the boolean test is required subsequently in the code.

4 The Monaco Run-Time System

Most procedural programming languages require little run-time support. A set of run-time libraries containing callable library functions are dynamically linked at load time or statically linked with the user program at compile time. Recent implementations of compiled object-oriented languages such as C++ provide run-time support for garbage collection. Concurrent logic languages, due to the nature of the process-oriented heap-based execution model, require extensive run-time support. We outline the main objectives of a run-time system below:

Process management

Computation in concurrent logic languages proceeds by selecting a goal for execution from a pool of ready-to-run goals. A goal reduction is carried out by executing the goal as a *light-weight* process (sometimes called a *thread*). A Monaco process may be thought of as a goal in execution. The process performs the clause try operations and if a clause commits, spawns child processes to represent the body goals of the clause. A goal does not always execute to completion. Often a goal will try to read the contents of an uninstantiated shared variable and will suspend. It is the responsibility of the goal that instantiates the variable to *wake* all the goals suspended on the variable and add them to the goal pool. Efficient light-weight process management is essential for high performance.

Heap management

Almost all processes need to allocate new heap memory in order to create data structures. Often the heap fills up and a garbage collection is needed to reclaim unused memory by collecting all the live heap data starting from a set of roots. All heap memory that is not reachable from any of the roots is collected as garbage. We did not implement garbage collection in this version of Monaco.

Performance evaluation

As discussed in Section 2, one of our main motivations in designing Monaco is to evaluate the effectiveness of our new abstract machine instruction set. The Monaco system on Sequent Symmetry is probably the first native-code implementation of FGHC on a commercial shared-memory multiprocessor. It is interesting to measure how much performance gain is achieved by native-code implementation compared with emulator-based implementations of similar languages, namely JAM Parlog and Panda. The run-time system needs to be instrumented for collecting statistics on execution times, heap usage, instruction usage counts, etc. This is discussed further in Section 6.

User interface and input/output

A programming environment in which source programs can be compiled, executed and debugged is required. Programs that require input/output need a mechanism to communicate with the external world.

All the previous design considerations have been addressed in the design of Monaco kernel, which is presented in this section.

4.1 The Monaco Kernel

The Monaco Kernel is a set of *worker* processes that run as regular Unix processes on all the processors. A worker process corresponds to one Monaco abstract machine processor. The number of processors can be specified when the system is invoked. We distinguish one of the host machine's processors as a *master* processor. This processor runs the *user* process which provides a simple user interface. The user process allows queries to be entered and the resulting bindings and statistics output. The master processor is also responsible for detecting termination of the computation and to initiate garbage collection (not yet implemented). We have also incorporated a small debugger in the run-time system which makes low-level debugging of parallel Monaco programs somewhat less painful. It can be used to set break points and inspect the abstract machine registers. The function of master and worker processes are shown in Figures 13 and 14, respectively.

4.2 Process Management

A worker process is normally in a *Hunt-Work* loop. It looks for work in the the goal pool, reduces it, adds the body goals of the clause to the goal pool, and looks for more work. The goal pool is implemented as a queue. The choice of queue discipline is an important factor in performance and will be discussed later in this section. If there was one central queue, access to it must be protected with a lock mechanism to prevent races. This would be a major performance bottleneck, effectively serializing the frequent operations of adding and removing goals from the goal queue. Thus we provide each processor with a private queue.

Goal queues are implemented with the `#PAIR` data type; the car part points to a goal record and the cdr part contains a pointer to the next list cell. Goals can be added to the queue at either end; but removed only from the front; There are two ways in which a process may be added to a run queue; at the front or at the rear of the queue. Adding new a new process at the front of the queue favors a *depth-first* search for the solution. Since processes are only removed from the front, the more recently added goals are

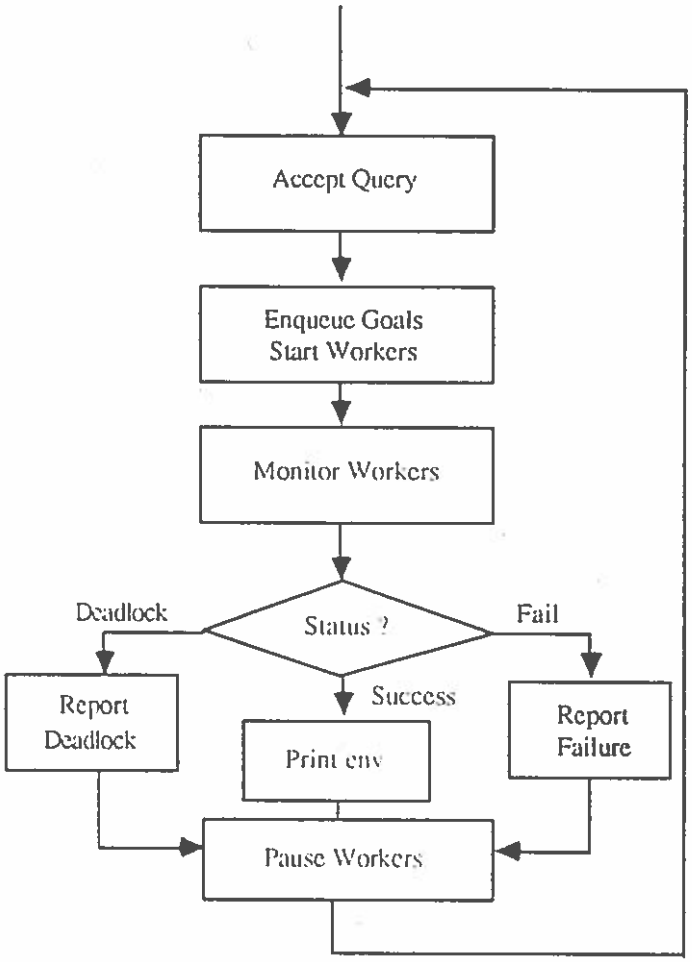

Figure 13: The Master Process

picked for execution before the goals enqueued earlier. Adding a goal at the rear of a queue favors a *breadth-first* search for the solution. Since committed-choice language programs find at most one solution, it is probably better to use depth-first style of search for the solution; but the choice of an enqueue policy appears to affect different benchmarks programs to varying degrees. Overall, the depth-first policy appeared to perform better. The same results were obtained by Sato [27].

Since each processor maintains a private queue, it enqueues all the new child goals that it creates, to the local queue. When a processor's goal queue becomes empty it needs to steal some work from a busy processor. We have adopted a work-search algorithm in which the idle processor peeks at the run queues of other processors in a circular fashion, starting with the right neighbor; stopping when the left neighbor has

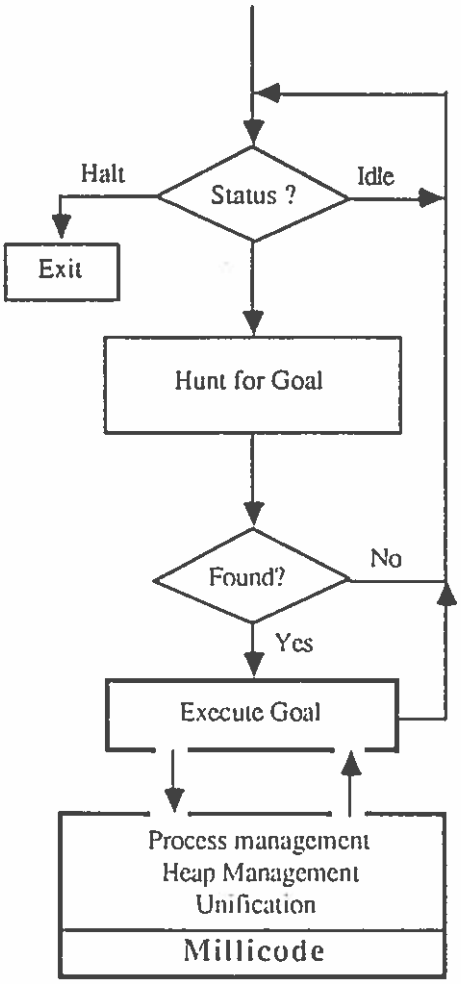

Figure 14: The Worker Process

been looked at. If work is found in any of the queues, the search is abandoned and the stolen goal starts executing immediately. This scheme is efficient because the idle processor is responsible for finding work, without involving the busy processors. This scheme works effectively and produces good load balancing from the speedups observed in multiprocessor execution. This method has one drawback in our implementation; since a goal queue is potentially accessed by multiple processors, access to the private goal queues also needs to be protected with a lock for each. Even when a processor needs to remove work from it's private queue, the lock has to be obtained. However, most of the lock operations will not block when all the processors are busy.

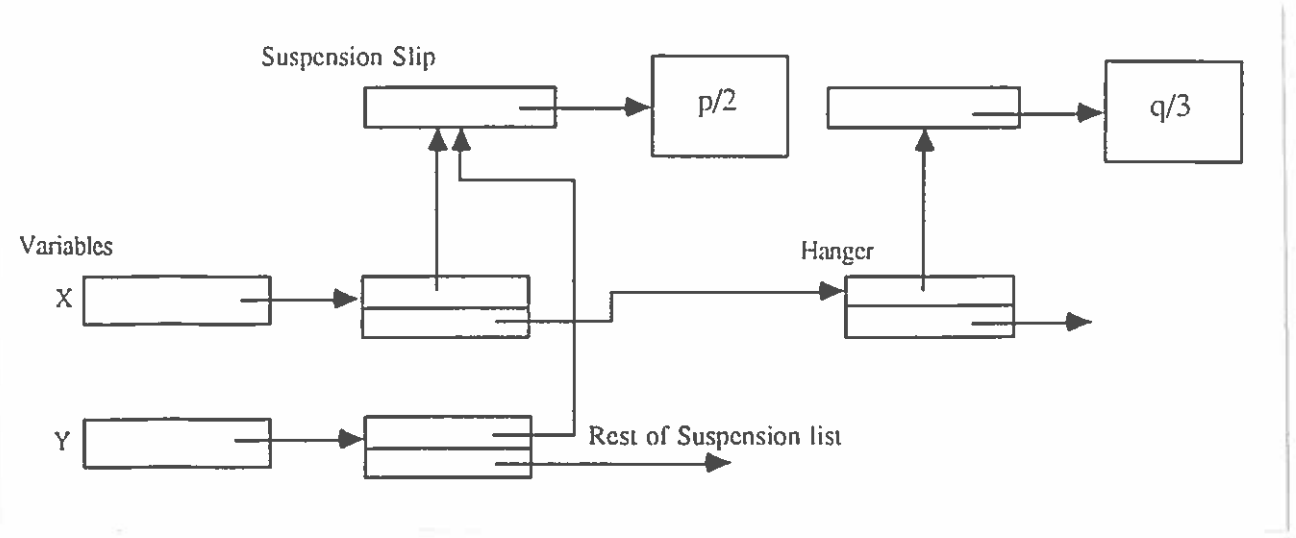

Figure 15: The Suspension Data Structures

4.3 Suspension and Resumption

Recall from the discussion on the execution model in Section 2, that a goal (which manifests itself as a *process*) may suspend when it tries to read an unbound variable. It is then added to the suspension list for that variable. Since the variable is potentially shared, there may be other processes concurrently suspending on the variable, thus creating a potential race. The **suspend** instruction is executed when all the clauses have been tried and none had committed. If the suspension stack is empty, then the clause try has failed and the computation is halted. However, if there are one or more variables on the suspension stack, the goal will suspend on all the variables placed on the suspension stack. Note that if any of the variables on the suspension stack got instantiated meanwhile, the **suspend** will abort and the goal would be placed on the goal queue. It is also possible that a goal suspends on more than one variable at the same time, i.e., resides on multiple suspension lists. The same technique is used in JAM Parlog [7] and KL1 [19] implementations.

When a variable that has goals suspended on it is instantiated, it is the responsibility of the instantiating process (the *writer*) to *wake* all the goals suspended on the variable, placing them on a goal queue. Since a goal may suspend on more than one variable simultaneously, care should be taken to see that the goal does not resume more than once. This is achieved by redirecting all the references to the suspended goal via a *suspension slip*. Figure 15 depicts the situation when goal `p/2` is suspended on two variables `X` and `Y`. Goal `q/3` is suspended only on `X`. Since `p/2` is in two suspension lists,

the suspension hanger for p in both the lists point to the suspension slip for p. The suspension slip for p contains a pointer to the goal p/2.

If Y is instantiated both p/2 and q/3 are resumed. Since p/2 is suspended on X as well, a null value is placed in the suspension slip for p/2 to indicate to a process instantiating X that p/2 has already been resumed.

4.4 Goal Record Reuse

When a goal suspends, the environment for the goal, i.e., its arguments are placed in a goal record and the goal record is enqueued. Oftentimes, due to the last call optimization, the process that executes a **suspend** instruction, and the process that was last dispatched by the run-time kernel are different. The kernel still probably has a reference to the goal record of the *ancestor* process, which can be reused for the suspending process, if the suspending process has an equal or smaller arity. If the arity of the suspending process is greater, goal record reuse is not possible. A free list of goal records for deferred reuse is possible, but has not been implemented.

4.5 Heap Management

Fast heap allocation improves the execution time of any program. The heap is initialized at start up with the heaptop pointer pointing to the first available heap cell and the heaplimit is set the maximum heap address. In each call to the memory allocator, the heaptop pointer is appropriately incremented and checked with the heaplimit for overflow. Since this operation occurs with great frequency, the heaptop and heaplimit are kept in the **globals** data structure, a reference to which is always available in the GLBL register.

Maintaining a single heap shared by all processors serializes the execution since the access to the heaptop pointer is then protected by a lock. We have split the heap into equal-sized slices for each processor. The split heap is effective because each processor can allocate memory on the heap in parallel, without lock synchronization. This is evidenced by the speedups achieved in multiprocessor execution. The technique of split heap was also implemented by Crammond in JAM Parlog [5].

Since each heap is privately managed, no synchronizing lock operations are required around heap allocation. So this operation is fairly inexpensive and we were able to expand it into fast native-code, thus avoiding function call overhead.

4.6 Millicode Procedures

Table 11 contains a list of the procedures implemented in millicode. These opera-

`unify (Rs1,Rs2)`	`active unify`
`punify (Rs1,Rs2,Rd)`	`passive unify`
`enqueue (Rs)`	`enqueue goal`
`suspend (Proc, Arity)`	`suspend goal`
`execute (Proc, Arity)`	`call Proc/Arity`

Table 11: Millicode Procedures

tions are quite complex, requiring a lot of synchronization and are better implemented efficiently in a high-level language such as C. Millicode procedures are dynamically linked to Monaco procedures via the millicode table. The millicode calling sequence is illustrated in Section 3.

4.7 Unification

Since our objective is to obtain high performance with simple abstract machine instructions, there is a single `unify` instruction that performs *active* unification and a `punify` instruction that performs *passive* unification. The unification algorithms use *dereference* operation to follow a chain of `#REF` pointers until a non-`#REF` type object is reached. We shall first present the *dereference* algorithm later followed by the *ActiveUnify* and *PassiveUnify* algorithms.

The data type `word` will be used in the algorithms to denote that the qualified object is expected to be a valid Monaco datum.

4.8 Dereference

The pseudo-code for the dereference operation is shown in Figure 16. The input `p` is a tagged Monaco datum. `CELLREF(x)` is the contents of the heap cell at location `x`.

All variables are potentially shared. If a chain of `#REF` pointers to a variable `X` is dereferenced, the dereference loop will terminate on encountering a non-`#REF` tag. If a `#VAR` or `#HOOK` tag was encountered, then `X` was not bound at the time the dereference terminated, but it may have been instantiated to some other data type, later. If the dereference operation terminated on encountering a value for e.g., `#PAIR`, the result of the dereference operation will always be the same, if the same chain of pointers was ever dereferenced again. In other words the `dereference` operation we have defined here does not guarantee the data type of the returned datum, if it was not bound to a value. For this reason we refer to the definition of dereference in Figure 16 as a *soft dereference*. To illustrate the problem consider the following scenario: variable `X` is

```
word SoftDereference(word p) {
  while (TAG(p) == #REF)
    p := CELLREF(p));
  return p;
}
```

Figure 16: The Soft Dereference Operation

```
word HardDereference(word p) {
 ObtainLock(p);
 while (TRUE) do {
    if (TAG(CELLREF(p)) == #REF) {
        ReleaseLock(p);
        p := CELLREF(p);
        }
    else { /* a non-#REF type datum */
     if ((TAG(CELLREF(p)) <> #VAR) and
         (TAG(CELLREF(p)) <> #HOOK)) {
         ReleaseLock(p);
         return CELLREF(p);
         }
     else
       return CELLREF(p);
    }
  }
}
```

Figure 17: The Hard Dereference Operation

shared between two goals `p(X)` and `q(X)`. Goal `p(X)` dereferences `X` and the result is a `#VAR` type datum. But goal `q(X)` may subsequently bind a value to `X` changing the type of `X` to some tag in the set `(#FIX,#IMM,#VEC,#PAIR)`, or it may suspend on `X` changing the tag to `#HOOK`. It is *unsafe* for `p(X)` to assume that `X` is still of type `#VAR`.

It is necessary to define a dereference operation that is stronger than *soft dereference*: the type of the returned datum must be guaranteed to hold. The extra certainty is obtained by locking the dereferenced object so that it cannot be changed after the dereference operation terminates. This operation is termed *hard dereference*, the pseudo-code given in Figure 17.

The hard dereference operation is always passed a `#REF` type datum. The object pointed to by `#REF` pointer is locked prior to checking the object's data type. One can visualize this as walking a chain of pointers, always locking the link just ahead before reading its contents. Since the lock of the object is obtained before the data type is checked, no other goal may change the data type of the locked object. If the result of dereference is a `#VAR` or `#HOOK`, the lock is retained and the dereference operation

returns to the caller. It is the responsibility of the caller to release the lock, after performing all the desired actions on the locked object. If the result of the dereference was any other data type, the lock is relinquished within the hard dereference operation, before returning to the caller.

The hard dereference operation can be expensive if long `#REF` pointer chains are dereferenced, because locking is required for every link in the chain. Excessive locking also retards the parallelism in programs. We have used the hard dereference operation only when it is absolutely necessary. There is also empirical evidence to show that the length of most `#REF` pointer chains is zero or one [32].

The actual soft dereference and hard dereference operations implemented in Monaco are almost identical to those presented here, with minor differences in handling `#HOOK`-type datum.

4.9 Variable Instantiation

Any Monaco variable is potentially shared between multiple goals and needs to be locked during binding. Many types of race conditions are possible as discussed below:

Multiple writers to a variable

> Assume that a variable `X` is shared between two goals `p(X)` and `q(X)`, and both the goals can instantiate `X`. To preserve the single-assignment property, all writers are required to check that the cell being written to is not instantiated. In order to prevent the two goals from instantiating `X` to two different values, write access to `X` must be serialized. This is accomplished by making all potential writers of `X` obtain the lock for `X` before `X` can be changed. This resolves races between multiple writers to a shared variable. This type of race does not occur very often since most FGHC programs are written so that a variable does not have multiple writers or *producers*. FGHC programs that do not have multiple producers for any variable are *properly moded*.

Suspending on a variable

> Assume that a variable `X` is shared between two goals `p(X)` and `q(X)`. One of them, say `p(X)`, tried to read `X`. Variable `X` was unbound and hence `p(X)` prepares to suspend on `X`, i.e., add itself to the suspension list for `X`. Goal `q(X)` meanwhile, instantiates `X`. Goal `q(X)` will resume any goals that are suspended on `X`. If we do not have some form of synchronization, then `p(X)` may never resume. A race condition also occurs when two processes are simultaneously suspending on the same variable. To prevent possible inconsistencies in the suspension-list data structure, suspend operations also must synchronize.

```
Instantiate(word X, word NewVal)
{
   ObtainLock(X).
   OldVal := X.
   X := NewVal
   ReleaseLock(X).

   if (OldVal is bound)
      unify(NewVal,OldVal)
   else
      if (goals suspended on X)
         wake goals on suspension list
}
```

Figure 18: Variable Instantiation Algorithm

Waking up a goal

Recall that a goal may suspend on multiple variables at the same time. Assume that a goal `p(X,Y,Z)` suspends on all the three variables `X`, `Y`, and `Z`. If any of the variables gets a value, then `p(X,Y,Z)` needs to be resumed by the instantiator of the variable. Since the suspended goal is on multiple suspension lists, potentially several instantiators are racing to wake `p(X)`. It is necessary to prevent the situation wherein a suspended goal is resumed more than once.

The algorithm used for instantiating a variable is shown in Figure 18. The instantiation algorithm was first described in [5]. `Instantiate(X,NewVal)` assigns the value `NewVal` to `X`. `Oldval` is the old value in `X`, if `X` was bound. This algorithm minimizes the time a heap cell is locked. If `X` was already bound, then `NewVal` is assigned to `X`, but it is verified that `OldVal` and `NewVal` are unifiable. Normally, `X` is not already bound and the `unify(NewVal,OldVal)` operation is not executed. Since synchronization is enforced by synchronizing the section of code where the old value of `X` is changed, races between multiple writers are eliminated.

4.10 Active Unification (`ActiveUnify`)

The `ActiveUnify` routine is invoked as `aunify(a,b)` where `a` and `b` are tagged Monaco objects. Both `a` and `b` are first dereferenced. The tags are checked to see if `a` and `b` are already instantiated. If both `a` and `b` are instantiated and `a` and `b` are simple types, i.e., `#FIX` or `#IMM`, a simple equality check determines if they are unifiable. If they are of `#PAIR` or `#VEC` type, then sub-structures must be recursively unifiable. If only one of `a` or `b` is unbound, the `Instantiate` algorithm described above, will be used. If both `a` and `b` are unbound, the unification is completed by making one of the variables point to

the other, i.e., one is converted to a #REF type pointer. We have adopted the following conventions to prevent circular reference chains from building up. They are based on address comparison which is possible on a shared-memory host.

1. `ActiveUnify(#VAR,#VAR)`

   When two variables are unified, the variable stored in the higher address is made to point to the variable stored at a lower address on the heap.

2. `ActiveUnify(#HOOK,#HOOK)`

   When two hooked cells are unified, the cell stored in the higher address is made to point to the hooked cell stored at a lower address and the suspension lists are concatenated.

3. `ActiveUnify(#HOOK,#VAR)` or `ActiveUnify(#VAR,#HOOK)`

   When a hooked cell is to be unified with a variable cell, the variable cell is always made to point to the hooked cell.

The algorithms are presented in C syntax. The implementation details are omitted for clarity. The top level of `ActiveUnify` is shown in Figure 19. The case where a #FIX datum is unified with a datum of undetermined type is shown in Figure 20. The case where a datum of type #PAIR, #IMM or #VEC is unified with a datum of unknown type is similar. Figure 21 contains the algorithm used when #VAR is unified with a datum of undetermined type. The algorithm for unifying with the #HOOK type datum is omitted as it is similar to the algorithm for #VAR, but with some extra complexity introduced for handling suspension lists.

On entry to `ActiveUnify(a,b)` (see Figure 19), a is soft dereferenced and the tag of a is checked to perform a type dispatch. If a is already instantiated, one of the specialized unification routines is called. If the tag of a is unbound, the hard dereference operation is performed to check if it is still unbound (see the description of hard dereference earlier in this section). The result of the hard dereference is reliable and is used to complete the top-level type dispatch. Note that hard dereference is an expensive operation because it involves locking. We use it only when absolutely necessary.

Figure 20 shows the unification algorithm when a is #FIX and b is not yet determined. First, b is dereferenced. If b is instantiated and is of type #FIX, a simple equality test determines the success of unification. If b is instantiated but to a different type, the unification fails. If b is unbound, the tag of b could have changed meanwhile, and hence b is locked and the tag of b is checked again. If b has been set to reference

```
int ActiveUnify(word a,word b)
{
  word tmp;

  a = dereference(a);

  switch (TAG(a)) {
  case FIX_TAG :
    return active_unify_fixnum(a,b);

  case PAIR_TAG :
    return active_unify_pair(a,b);

  case IMM_TAG :
    return active_unify_imm(a,b);

  case VECTOR_TAG :
    return active_unify_vector(a,b);

  case VAR_TAG  :
  case HOOK_TAG :

    a = HardDereference(VALUE(a));

    switch (TAG(a)) {
    case IMM_TAG :
      return active_unify_imm(a,b);
    case FIX_TAG :
      return active_unify_fixnum(a,b);
    case VEC_TAG :
      return active_unify_vector(a,b);
    case PAIR_TAG :
      return active_unify_pair(a,b);
    case VAR_TAG :
      return active_unify_var(a,b);
    case HOOK_TAG :
      return active_unify_hook(a,b);
    default :
      return AUNIFY_FAILURE;
    }

  case REF_TAG :
    return ActiveUnify(a,b);

  default :
    return AUNIFY_FAILURE;
  }
}
```

Figure 19: The Top Level of ActiveUnify

```
int active_unify_fixnum(word a, word b)
{   word oldb;
    Tag  newTagOfb;

    b = dereference(b);
    switch (TAG(b)) {
    case FIX_TAG : if (a == b)
                      return AUNIFY_SUCCESS;
                   else
                      return AUNIFY_FAILURE;
    case IMM_TAG  :
    case VEC_TAG  :
    case PAIR_TAG : return AUNIFY_FAILURE;

    case VAR_TAG  :
    case HOOK_TAG : b = VALUE(b);
                    ObtainLock(b);

                    newtTagOfb = TAG(CELLREF(b));
                    if (newTagOfb != REF_TAG) {
                        oldValOfb = CELLREF(b);
                        Assign(b,a);
                    }
                    else {
                      ReleaseLock(b);
                      return ActiveUnify(a,b);
                    }

                    switch (newTagOfb) {
                    case VAR_TAG : ReleaseLock(b);
                                   return AUNIFY_SUCCESS;

                    case HOOK_TAG: WakeGoals(oldValOfb);
                                   ReleaseLock(b);
                                   return AUNIFY_SUCCESS;

                    case FIX_TAG : ReleaseLock(b);
                                   if (a == oldb)
                                      return  AUNIFY_SUCCESS
                                   else
                                      return  AUNIFY_FAILURE;
                    case IMM_TAG  :
                    case PAIR_TAG :
                    case VEC_TAG  : ReleaseLock(b);
                                    return AUNIFY_FAILURE;

                          default : ReleaseLock(b);
                                    return AUNIFY_FAILURE;
                    }
      case REF_TAG : return ActiveUnify(a,b);
           default : return AUNIFY_FAILURE;
   } }
```

Figure 20: Algorithm for `ActiveUnify(#FIX,?)`

some other datum, the unification is retried. Otherwise, b is instantiated in the manner described in the `Instantiate(X,NewVal)` algorithm.

Figure 21 shows the unification algorithm, when the type of a is `#VAR` and a is locked. Input b is soft dereferenced and its tag is checked. The interesting case occurs when b is of type `#VAR`. Note that there would be a potential deadlock situation if we try to lock b immediately on discovering that it is `#VAR`. The deadlock occurs if two processes are concurrently performing `ActiveUnify(a,b)` and `ActiveUnify(b,a)`. One process locks a and is waiting to lock b. The other process locks b and is waiting to acquire the lock for a. The deadlock is avoided by requiring that higher addresses point to lower address. Since locking is done after the address comparison, deadlock is avoided. If b changed to `#REF`, i.e., set to reference some other variable meanwhile, the locks on both a and b are released and the unification is tried again. If b is of type `#HOOK`, a is required to reference b by our convention. In other cases where b is instantiated to some value, the value contained in b is copied into a.

4.11 Passive Unification (`PassiveUnify`)

The `PassiveUnify` routine is invoked as `punify(a,b)` where a and b are tagged objects. The algorithm for passive unification is given in Figure 22. Both a and b are dereferenced. If both a and b are dereferenced to the same value, the passive unification succeeds immediately. Otherwise, the tags of a and b are checked to see if either of them is unbound, i.e., `#VAR` or `#HOOK`. If either of a and b, or both a and b are unbound variables, then the unbound variable(s) are pushed on the suspension stack. Since the passive unification could not be completed due to unbound variables, the result `PUNIFY_SUSPEND` is returned.

After performing the above mentioned checks, the only other alternative is that both a and b are bound, and they *cannot* be simple type, i.e., `#FIX` or `#IMM`, because that would have been detected in the equality check performed at the beginning. A tag equality check is then made to see that both and a and b are of the same type. The passive unification is then completed by recursively performing passive unification on the head and tail in case of `#PAIR`. In case of `#VEC` the arities of a and b are checked for equality. If the arity check succeeds passive unification is performed on the corresponding vector slots of a and b. On success `punify(a,b)` returns `PUNIFY_SUCCESS`. The result `PUNIFY_FAILURE` is returned on failure.

4.12 Tuning the Algorithms

The algorithms were initially tuned by avoiding function calls inside the unify routine, keeping loop indices in registers, and programming "early" evaluation of simple con-

```
int active_unify_var(word a, word b)
{
  a = VALUE(a);       /* a is a pointer to Locked #VAR cell */
  b = dereference(b);
  switch (TAG(b)) {
  case VAR_TAG  :
        b = VALUE(b);      /* Unify(VAR,VAR?)              */
        if (a > b) {       /* AddressOf(a) > AddressOf(b) */
           Assign(a,b);    /* a REFs b     */
           ReleaseLock(a);
           return AUNIFY_SUCCESS;
        }
        else if (a < b) {        /* b needs to REF a   */
                ObtainLock(b);{  /* Peek at TAG again  */
                switch (TAG(CELLREF(b)))
                case REF_TAG : if ((CELLREF(b)) == a) { /* b REFs a */
                                    ReleaseLock(a); ReleaseLock(b);
                                    return AUNIFY_SUCCESS;
                               }
                               else { /* b REFs ? */
                                  ReleaseLock(a); ReleaseLock(b);
                                  return do_unify(a,b);
                                       }
                 case VAR_TAG : Assign(b,a); /* b still VAR */
                                ReleaseLock(a); ReleaseLock(b);
                                return AUNIFY_SUCCESS;
                 case HOOK_TAG: Assign(a,b); /* b is HOOK. a REFs b */
                                ReleaseLock(b); ReleaseLock(a);
                                return AUNIFY_SUCCESS;
                 case IMM_TAG :
                 case FIX_TAG :
                 case VEC_TAG :  /* b is *not* unbound */
                 case PAIR_TAG :  Assign(a,CELLREF(b))
                                  ReleaseLock(a); ReleaseLock(b);
                                  return AUNIFY_SUCCESS;
                 }
              }
      else {
         ReleaseLock(a);
         return AUNIFY_SUCCESS;
         }
  case FIX_TAG  :
  case IMM_TAG  :
  case VEC_TAG  :
  case PAIR_TAG : Assign(a,b); /* b is *not* unbound */
                  ReleaseLock(a);
                  return AUNIFY_SUCCESS;
  case HOOK_TAG : Assign(a,VALUE(b)); /* b is #HOOK. a REFs b  */
                  ReleaseLock(a);
                  return AUNIFY_SUCCESS;
       default : return AUNIFY_FAILURE;
  } } }
```

Figure 21: Algorithm for `ActiveUnify(VAR,?)`

```
int PassiveUnify(word a, word b)
{

  a = dereference(a);
  b = dereference(b);

  if (a == b)
     return PUNIFY_SUCCESS;

  TagOf_a = TAG(a); TagOf_b = TAG(b);

  if (TagOf_a == #HOOK) or (TagOf_a == #VAR)
      PushOnSuspensionStack(a);
  if (TagOf_b == #HOOK) or (TagOf_b == #VAR)
      PushOnSuspensionStack(b);

  If (SuspensionStackPtr > 0)
       return PUNIFY_SUSPEND;

  if (TagOf_a != TagOf_b)
       return PUNIFY_FAILURE;

  switch (TagOf_a) {
         case FIX_TAG  :
         case IMM_TAG  : return PUNIFY_FAILURE;

         case PAIR_TAG : car = do_punify(CAR(a),CAR(b));
                         if (car <> PUNIFY_SUCCESS)
                            return car;
                         cdr = punify(CDR(a),CDR(b));
                         return cdr;

         case VEC_TAG :  LengthOf_a = VECTORLENGTH(a);
                         LengthOf_b = VECTORLENGTH(b);

                         if (LengthOf_a != LengthOf_b)
                             return PUNIFY_FAILURE;

                         for (i=1; i <= LengthOf_a; i++) {
                               slot = punify(VECTORREF(a,i),
                                             VECTORREF(b,i));
                               if (slot != PUNIFY_SUCCESS)
                               return slot;
                         }
                         return PUNIFY_SUCCESS;

            default  :   return PUNIFY_FAILURE;
  }
}
```

Figure 22: The Passive Unification Algorithm

ditions by which the result of unification can be decided quickly, whenever possible. Also, locks are never acquired unnecessarily and never held for a long period, as this reduces parallelism. The active unification algorithm needed approximately 700 lines and the passive unification algorithm needed approximately 80 lines of C source code.

The `ActiveUnify` algorithm is quite complex and is hard to optimize. Through proper instrumentation, the most frequently-used paths through the unify algorithm can be identified. These critical sections of code will need performance tuning or implementation in native code.

Our intuition is that `ActiveUnify` will have only a few heavily-used sections for performing unifications; for example in the limited number of benchmarks we had studied, `ActiveUnify(#VAR,#PAIR)` is the the most frequently used call pattern to `ActiveUnify`. Collecting statistics of call patterns is one of the objectives of performance evaluation, described in the next section.

We conjectured that `PassiveUnify` is most likely to be called to check equality of two integers or atoms. Note that the equality check, immediately after the first dereference in `PassiveUnify` algorithm, performs can complete the passive unification quickly.

5 Performance Evaluation

Preliminary performance evaluation of Monaco has focussed on measuring the uniprocessor execution time, speedups in parallel execution, and profiling of program execution. All three aspects are discussed in this section. For uniprocessor execution, Monaco is compared with JAM [7] and PDSS [19]. For multiprocessor execution, comparison is made only with JAM. The initial evaluation indicates that our instruction set is a viable alternative to the traditional WAM-style instruction sets used in these systems.

5.1 Evaluation

Monaco is specialized for parallel execution of FGHC programs. It is implemented in native code, with the exception of process management and unification which are implemented in millicode. For a description of the millicode mechanism, see Section 4.

JAM Parlog is a highly efficient emulator implementation of Jim's Abstract Machine for Parlog [7]. It includes support for *deep* guards. This introduces additional overheads for process management, since a process activation tree must be maintained for deep guards. JAM uses specialized unify instructions. The Parlog compiler also uses explicit mode declarations to generate efficient code. A parallel garbage collector is implemented in JAM.

PDSS is an emulator of KL1-B instruction set for FGHC [19]. It has specialized unify instructions, and uses the MRB incremental garbage collection scheme to dynamically reclaim memory [4].

We have chosen four small benchmarks for measuring the performance: *hanoi*, *nrev*, *qsort*, and *queens*. The benchmarks exhibit varying degrees of parallelism. The *hanoi* program solves the Towers of Hanoi problem. *nrev* is the naive reverse program. *qsort* is the Quicksort program with a presorted input. *qsort(r)* is the same as *qsort*, but with the input in reverse sorted order. We have observed some interesting differences in the behaviors of Monaco, JAM, and PDSS depending on whether the input list is in sorted or reverse sorted order. *queens* solves the n-queens problem. These benchmark programs for evaluating parallel logic programming systems may be found in [33].

All the benchmarks have been executed on the same configuration, a Sequent Symmetry S81 system with 16 MHz Intel 386 microprocessors. The execution time is the elapsed time between starting the computation, until all the processors become idle. All times are in milliseconds.

Program	Monaco(ms)	JAM(ms)	J/M	PDSS(ms)	P/M
hanoi	331	455	1.4	580	1.8
nrev	37,360	38,465	1.0	55,580	1.5
qsort	9,106	13,055	1.4	15,200	1.7
qsort(r)	9,268	16,490	1.8	20,090	2.2
queens	53,596	140,400	2.6	179,060	3.3

Table 12: Uniprocessor Performance of Monaco Compared to JAM and PDSS

5.2 Uniprocessor Performance

Table 12 shows the uniprocessor performance of each benchmark on Monaco, JAM, and PDSS. The J/M column shows the ratio of the execution time on JAM to the execution time on Monaco. The P/M column shows the ratio of execution times on PDSS and Monaco.

The uniprocessor execution times on Monaco are often better than that on either JAM or PDSS, and never worse. In the case of *queens* it is 2.6 times faster than JAM and 3.3 times faster than PDSS. The execution times on JAM may have been affected by the overhead of garbage collection. The heap size in JAM Parlog V1.4 is a function of the number of processors. For uniprocessor execution, the heap size is small and causes frequent garbage collection for memory intensive programs [6].

The *qsort* and *qsort(r)* illustrate the superiority of the decision graph algorithm used in Monaco compiler for clause indexing. The execution times for both *qsort* and *qsort(r)* are nearly the same on Monaco. It is desirable that a program's execution time be independent of the ordering of elements in input list, if the size of the input is the same. The execution time for *qsort(r)* is more than the time for *qsort*, for both JAM and PDSS. We conjecture that this difference is due to the ordering of the clauses in the *partition/4* procedure which partitions the input list into two sub lists, containing elements smaller and larger than the pivot element. Clause indexing in JAM and PDSS appears to be more expensive than in Monaco.

5.3 Multiprocessor Performance

The multiprocessor execution times are given in Table 13. The ratio between Monaco and JAM parallel execution times is given in Table 14. The speedups are shown in Table 15. The speedup is calculated as the ratio of uniprocessor and multiprocessor execution speed. This metric is not accurate, since the the overhead of support of parallelism stifles uniprocessor execution speed, resulting in slightly exaggerated speedups. So the speedup is not an absolute measure but it is a good indicator of multiprocessor

Program	1 PE	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco(ms)					
hanoi	331	170	94	54	41	41
nrev	37,360	20,782	10,456	5,367	3,671	2,845
qsort	9,100	5,034	2,596	1,334	906	704
qsort(r)	9,236	5,175	2,675	1,382	952	740
queens	53,596	49,562	15,919	7,569	4,899	3,689
	JAM(ms)					
hanoi	455	185	105	35	30	32
nrev	38,210	34,970	9,985	4,880	2,890	2,320
qsort	13,055	6,756	6,385	4,530	4,683	2,633
qsort(r)	16,110	7,800	3,620	1,990	1,770	1,448
queens	140,100	69,810	35,190	17,470	11,730	5,823

Table 13: Multiprocessor Performance of Monaco and JAM

Program	2 PE	4 PE	8 PE	12 PE	16 PE
	JAM/Monaco				
hanoi	1.1	1.1	0.7	0.7	0.8
nrev	1.7	1.0	0.9	0.8	0.8
qsort	1.3	2.5	3.4	5.2	3.8
qsort(r)	1.5	1.4	1.4	1.9	2.0
queens	1.4	2.2	2.3	2.4	1.6

Table 14: Multiprocessor Execution: Monaco Compared to JAM

Program	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco				
hanoi	1.9	3.5	6.1	8.1	8.1
nrev	1.8	3.6	6.9	10.2	13.1
qsort	1.8	3.5	6.8	10.0	12.9
qsort(r)	1.8	3.5	6.7	9.7	12.5
queens	1.1	3.4	7.1	10.9	14.5
	JAM				
hanoi	2.3	3.8	13.0	15.2	15.2
nrev	1.1	3.8	7.8	13.1	16.5
qsort	1.9	2.0	2.8	2.8	4.9
qsort(r)	2.1	4.5	8.1	9.1	13.1
queens	2.0	4.0	8.1	12.0	24.1

Table 15: Multiprocessor Speedups on Monaco and JAM

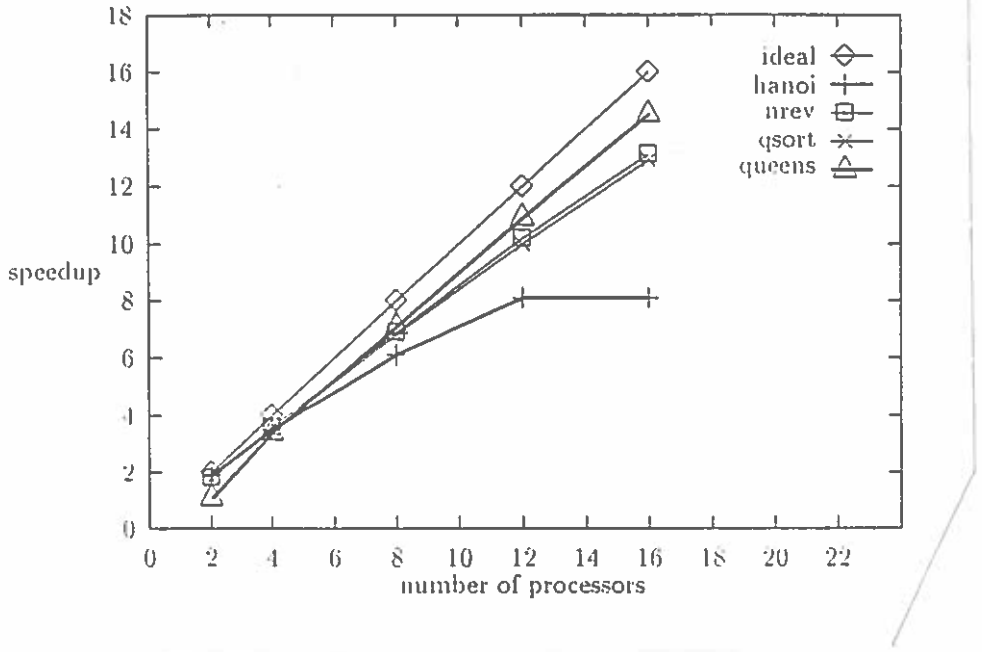

Figure 23: Multiprocessor Execution: Speedups on Monaco

utilization.

Note from Table 14, that the speedups on Monaco are usually better than that of JAM, but sometimes worse. We conjecture that the difference is due to the locality of memory references and better cache utilization in JAM, which was also reported by Tick and Crammond in [35]. JAM uses 32-bit data resulting in better cache utilization. JAM uses an *argument stack* where the arguments to a procedure are placed. The stack is often compacted leading to more efficient use of memory. The goal records are of a fixed size. Both Monaco and JAM used almost identical scheduling algorithms, except that an idle Monaco worker process will steal work from the front end of a busy worker's queue, instead of the rear end, as in JAM. Furthermore, the measurements on JAM had a high variance. We report the average over several runs of a benchmark. The variance was as high as 800% in some cases. We assume this was due to garbage collection.

The speedups obtained on Monaco are shown graphically in Figure 23. The *hanoi* program does not have a lot of parallelism. The speedup for *hanoi* increases until the number of processors reaches 12, and then tapers off. The *nrev*, *qsort* and *queens* are highly parallel and good speedups are achieved even when the number of processors is increased to 16. Note that the performance of all the three benchmarks improves smoothly with addition of processors. This indicates that our design and choice of the scheduling algorithm produces good load balancing. It also shows that there are no

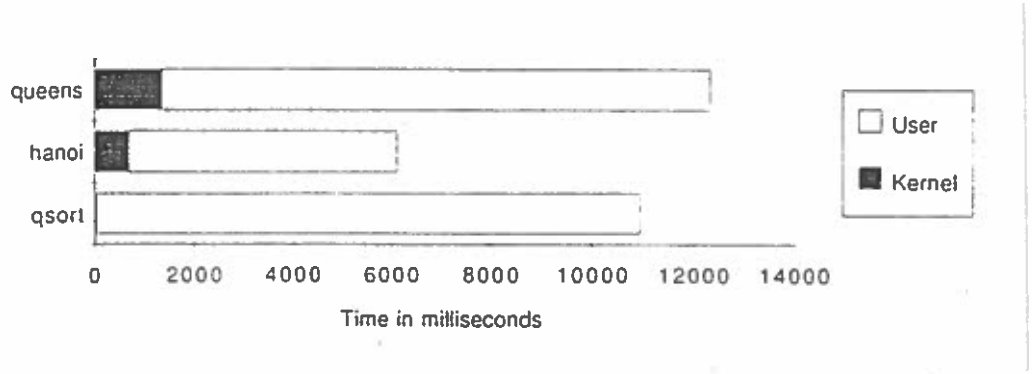

Figure 24: The Execution Times in Kernel and User Program

bottlenecks in the system. Programs that have a high degree of inherent parallelism, parallelize well on the Monaco system.

JAM also showed good speedup in multiprocessor execution for all the benchmark programs. Some of the JAM measurements showed *superlinear* speedups. This phenomenon is partly due to the fact that speedup calculation was based on the uniprocessor execution time of a parallel system and not on the execution time of the best sequential system. Since programs written for parallel execution may not execute as efficiently as programs written especially for uniprocessor execution, the observed speedups for JAM, as well as Monaco may be slightly exaggerated. More importantly, JAM allocated a smaller heap in uniprocessor execution, resulting in greater frequency of garbage collection, and inflated uniprocessor execution times[6]. The above two factors are mainly responsible for the superlinear speedups observed with JAM.

The different behavior of *qsort* and *qsort(r)* which was observed in uniprocessor execution manifested itself more adversely, in multiprocessor execution. Monaco achieved consistently better speedups, the best performance being, over five times faster than JAM with 12 processors.

5.4 The Execution Profile

We instrumented the Monaco kernel to measure the performance of the frequently-used millicode procedures, namely `unify`, `enqueue`, and `suspend`. All the measurements were made with uniprocessor execution. The profiled programs are *qsort*, *hanoi*, and *queens*. The overhead of profiling was as high as 20% for *qsort*. The inputs to benchmarks were chosen so that profile data is collected over a reasonably long period of execution. Figure 24 shows the time spent in the run-time system and in the program. The total execution time is the sum of times spent in the user program and the run-time

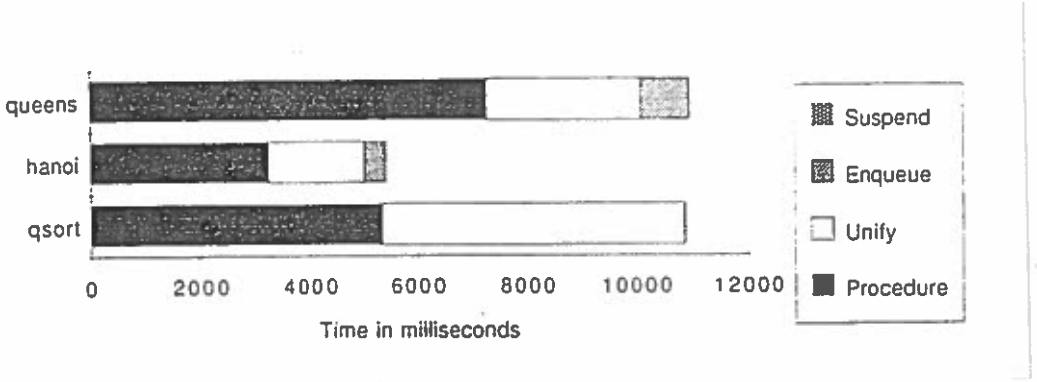

Figure 25: The User Program Execution Profile

Program	%Time	No. Calls	Avg Time(μs) per Call
hanoi	32	32,768	53
qsort	50	132,866	41
queens	25	10,897	60

Table 16: The Performance of Unification

kernel, it is shown along the X-axis. The time spent in the Monaco kernel is a small percentage of the total execution time.

The time spent in the user program includes the time for setting up the arguments in the abstract machine registers and executing the user program. The user program execution time can be split up into the following four major components: in the millicode procedures for **unify**, **enqueue**, and **suspend**, and in the *rest* of the procedure body.

Figure 25 shows the time spent in the user program, split into the times spent in each of the above four activities. A negligible amount of time was spent in **suspend**, for these benchmarks. The *hanoi* and *queens* benchmarks enqueue a large number of goal records and hence the time spent in the millicode procedure **enqueue** is significant. The time spent in the **unify** millicode procedure varied for each program. This indicates that optimizing unification will improve performance of programs to varying degrees. A major percentage of the user program execution is spent in the procedure body, and this percentage may improve further, when the millicode procedures are optimized.

The unification performance in *hanoi*, *qsort*, and *queens* was analyzed, as presented in Table 16. The **%Time** column shows the time spent performing unification as a percentage of the total execution time. The **No. Calls** column shows the number of calls to the **unify** procedure. The last column shows the average time per call in

	Monaco	JAM	PDSS
Abstract machine instructions	627	?	498
Size of object/bytecode file	13,320	3,437	1,664

Table 17: Static Characteristics of Code in Monaco, JAM and PDSS

microseconds.

The three programs we analyzed, predominantly manipulate lists. In the *qsort* program, over 98% of the calls unify a `#PAIR` type datum with a `#REF` type datum in the *partition/4* procedure. The *hanoi* and *queens* programs spend a relatively smaller percentage of time performing unification, though the average cost of an unification call is higher in both programs. The *queens* program spent the smallest percentage of its execution time performing unification. A larger percentage is spent in performing the complicated guard tests. This indicates that specializing unification will not make *queens* execution much faster.

These measurements were made with a system that was largely untuned. With tuning, the average cost of unification is expected to reduce considerably. We have not yet implemented the *assign* instruction. When it is implemented, it may replace some of the calls to `unify`.

5.5 Analysis of Generated Code

The Monaco compiler generates a `*.mon` file containing Monaco abstract machine instructions. The Monaco assembler generates intel 386 assembly language instructions from the `*.mon` file, producing a `*.s` file. The `*.s` file is then assembled using the standard Unix assembler *as*, producing a `*.o` file. The Monaco runtime system is then statically linked with the `*.o` file. The Monaco runtime system is approximately 74KB in size before linking with our current set of benchmarks. After linking, the size of the Monaco executable file is approximately 84 KB.

We used Parallel Parlog V1.4 for our JAM benchmarks. The JAM compiler generates an object file containing bytecodes for the JAM instruction set. JAM does not generate a file containing readable abstract machine instructions. The JAM runtime system is approximately 70 KB in size.

We used PDSS-KL1 V2.52.19 for our measurements. The KL1 compiler generates two files, a `*.asm` file containing KL1-B abstract machine instructions, and a `*.sav` containing the bytecodes for abstract machine instructions. The PDSS system itself is 384 KB in size. A large part of the code size is support for MRB garbage collection.

Table 17 compares the code sizes produced by the three systems. Note the almost

four-fold increase in the object file size, as a result of native-code compilation.

6 Conclusions and Future Work

We have completed the first implementation of the Monaco abstract machine on Sequent Symmetry. We have performed preliminary performance benchmarking and the results indicate that our instruction set and native-code implementation result in good uniprocessor performance when compared to JAM and PDSS. We have also measured the multiprocessor performance and found that programs which have a good amount of parallelism achieve near-linear speedups. We have compared the multiprocessor execution times on Monaco with the times on JAM and find that the performance is comparable and sometimes better.

6.1 What Did We Learn?

We learned that it is worthwhile to re-explore the design space for instruction sets in concurrent logic programming. We have demonstrated that the Monaco abstract machine, although unconventional, is a viable design. The performance is comparable to that of traditional designs.

The simplicity of the abstract machine instructions reduced the implementation time. The abstract machine currently has 58 instructions of which only five instructions are sufficiently complex to be implemented in millicode. The rest were easily translated into short assembly instruction sequences. The system was developed and debugged in a short period mainly because of the simple and orthogonal abstract machine design. Compiling to native-code was not as hard as it first appeared to be. Since we do not have specialized unification instructions, the writing and debugging of the compiler was simplified to some extent. The complexity of the compiler was largely in implementing optimizations such as decision graph, code generation, and data flow analysis.

We also learned that the intel 386 processor is far from an ideal target for the Monaco abstract machine. The intel 386 has too few general-purpose registers. None of the abstract machine registers could be mapped to the host processor's registers. We could only map references to some critical run-time system data structures onto the host processor's registers. The shortage of registers was also acutely felt when we attempted to pass arguments to millicode procedures through registers. We did not see scope for many interesting peephole optimizations on the generated native code.

6.2 Future Work

Tune the Run-Time System

Note that the results we have obtained thus far are achieved after performing simple performance tuning, such as reducing the number of millicode calls. We

have demonstrated that the performance and speedup of Monaco are comparable to JAM, in the first attempt. We believe that with further tuning, the Monaco system can be improved significantly. We have profiled the system to measure the frequencies and the time taken to perform critical activities such as unification, suspension and resumption. We plan to use the gathered statistics to guide our efforts towards optimizations with maximum pay-off. ActiveUnify and PassiveUnify algorithms are presently largely unoptimized and are immediate targets for optimization efforts.

Improve the Compiler

Presently the compiler performs some traditional optimizations such as common subexpression elimination, register allocation, etc. As we discussed in Section 2, the compiler can detect where assignment suffices for unification. But the mode analysis [23] required for this optimization has not yet been incorporated in the compiler.

Other Scheduling Strategies

We have used a simple queuing organization for goal queue management. The queuing discipline favors depth-first search and achieves good performance for our limited number of benchmarks. We need to experiment with other queue management schemes by applying results from mode analysis to keep producers executing ahead of consumers. This will reduce the frequency of suspension, which is an expensive operation.

A RISC Implementation

Any of the newer pipelined RISC architectures would make a better target for our abstract machine. The large register files of RISC architectures would allow some of the abstract machine registers to be kept in the target processor's registers. Calling millicode procedures would be less expensive because arguments and return address could be passed through registers. We plan to port the Monaco system to a RISC multiprocessor architecture.

Specialize Unification

We do not rule out the possibility of having specialized unify instructions in the future, in order to measure how much difference they make in execution efficiency.

Memory Reuse

Better locality in memory references can be obtained by memory reuse schemes. Limited reuse of goal records has been implemented in Monaco (see Section 4).

We plan to experiment with free lists for full reuse of goal records. It is possible that the run-time overhead may offset any improvements in execution time. General reuse of heap data structures is difficult for concurrent logic languages. A reference-counting approach can be tried, since there is space available in the Monaco datum to incorporate a reference count. However, compile-time detection of garbage cells is more attractive because the price of detecting garbage cells is paid only once, at compile time. Constructing efficient algorithms for compile-time garbage collection has received attention lately and some of the benefits have been demonstrated[13, 10]. We hope to implement memory reuse in the near future.

A Semantics of Monaco Abstract Machine

The semantics of the Monaco Abstract Machine instruction set is defined in this Appendix.

A.1 Data Constructors

Data constructors provide a means to create data structures and values in an implementation independent fashion. We do not provide constructors for all data types that the abstract machine knows.

`mkconst (<const>, Rd)`

An appropriately tagged literal is placed in `Rd`. `<const>` is any valid constant in the source language.

`mklist (Rs1, Rs2, Rd)`

`Rs1` is the head of the new cell, and `Rs2` is the tail. A list cell is created on the heap, and a tagged pointer to the newly created cell is placed in `Rd`.

`mkstruct (Size, Rd)`

`Size` is the size of the vector, which must be a positive integer. A vector of size `Size` is created on the heap and a pointer to the newly created vector is placed in `Rd`.

`mkgoal (Size, Proc, Rd)`

Constructs a goal record with `Size` slots. `Size` must be a positive integer. `Proc` is a procedure name. The first slot is filled with a pointer to the procedure `Proc`. A pointer to the goal record is placed in `Rd`.

`mkvar (Rd)`

Creates a variable cell on the heap and places a `#REF` pointer to it in the `Rd` register.

`alloc (n, Rd)`

Allocates `n` cells on heap. Sets contents of `Rd` to hold a `#REF` pointer to first cell of the newly allocated block.

`ref (Rs, n, Rd)`

Register `Rs` contains a `#REF` pointer to a reserved block of `n` cells on the heap, obtained through a prior `alloc` instruction. The offset `n` is added to this heap

address, and the contents of Rd is set to contain a #REF pointer to this new address.

`initvar (Rd)`

Similar to the `mkvar` instruction, except that Rd already contains a #REF pointer to a free cell on the heap. The heap cell is initialized with a self pointer and #VAR tag. The tag of Rd is not changed.

`initlist (Rs1, Rs2, Rd)`

Similar to the `mklist` instruction, except that Rd already contains a #REF pointer to a free cons cell on the heap. The tag of Rd is changed to #PAIR. Contents of registers Rs1 and Rs2 are copied into the head and tail positions.

`initstruct (Size, Rd)`

Similar to the `mkstruct` instruction, except that Rd already contains a #REF pointer to a frame on the heap. The header word of the vector frame is set to indicate that it represents a structure. The tag of Rd is changed to #VEC. The size of the vector is set to Size.

`initgoal (Size, Rd)`

Similar to the `mkgoal` instruction, except that Rd already contains a #REF pointer to a free vector frame on the heap. The header word of the vector frame is set to indicate that it represents a goal record. The tag of Rd is changed to #VEC. The size of the vector is set to Size.

A.2 Data Manipulators

Data manipulators access and update data structures. In general, the type of the data structure being updated is encoded in the pointer to the structure, and hence type checking can be done. However, range checking is not performed in general.

`deref (Rs, Rd)`

Soft dereference the contents of Rs and place the result in Rd. We use the term *soft* to indicate that `deref` will not lock the result, and if the result is an unbound its tag may change after `deref` is performed.

`move (Rs, Rd)`

Copy register Rs into register Rd.

`sset (Rs, n, Rd)`

Register `Rd` must be a pointer to a vector. The n^{th} slot of the vector is set to the contents of register `Rs`. If `Rd` is not a vector pointer, then an exception is raised. The value of `n` is assumed to be within range for the particular vector.

`sref (Rs, n, Rd)`

Register `Rs` must be a pointer to a vector. The n^{th} slot of the vector is fetched and put into register `Rd`. If `Rs` is not a vector pointer, then an exception is raised. The value of `n` is assumed to be within range for the particular vector.

`ssize (Rs, Rd)`

Register `Rs` must be a pointer to a vector. The arity of the vector is returned as a tagged integer in register `Rd`.

`car (Rs, Rd)`

Register `Rs` must be a pointer to a list cell. The head of the list cell is fetched and put into register `Rd`. If `Rd` is not a pointer to a list cell, an exception is raised.

`cdr (Rs, Rd)`

Register `Rs` must be a pointer to a list cell. The tail of the list cell is fetched and put into register `Rd`. If `Rd` is not a pointer to a list cell, an exception is raised.

The following instructions are currently not emitted by the compiler. They are intended to facilitate resue of list cells when the structure-reuse analysis is implemented in the compiler.

`setcar (Rs, Rd)`

Register `Rd` must be a pointer to a list cell. The head of the list cell is replaced by the value of register `Rs`. If `Rd` is not a pointer to a list cell, an exception is raised.

`setcdr (Rs, Rd)`

Register `Rd` must be a pointer to a list cell. The tail of the list cell is replaced by the value of register `Rs`. If `Rd` is not a pointer to a list cell, an exception is raised.

A.3 Predicates

A predicate leaves a truth value in its destination register; the truth value is a tagged integer, with `FIX#1` representing true and `FIX#0` representing false. Type predicates

isint (Rs, Rd)	tagof(Rs) == #FIX ? Rd := True : Rd := False
isref (Rs, Rd)	tagof(Rs) == #REF ? Rd := True : Rd := False
isunbound (Rs, Rd)	tagof(Rs) == #VAR ? Rd := True : Rd := False
ishooked (Rs, Rd)	tagof(Rs) == #HOOK ? Rd := True : Rd := False
islist (Rs, Rd)	tagof(Rs) == #PAIR ? Rd := True : Rd := False
isnil (Rs, Rd)	Rs == [] ? Rd := True : Rd := False
isstruct (Rs, Rd)	tagof(Rs) == #VEC ? Rd := True : Rd := False
isimm (Rs, Rd)	tagof(Rs) == #IMM ? Rd := True : Rd := False
eq (Rs1, Rs2, Rd)	Rs1 == Rd2 ? Rd := True : Rd := False
neq (Rs1, Rs2, Rd)	Rs1 == Rd2 ? Rd := False : Rd := True

Table 18: Type Predicates and General Predicates

and general predicates are shown in Table 18. Type predicates are used to check the tags of tagged words. Integer predicates are shown in Table 19; if an integer predicate is passed a non-integer, it raises an exception.

Table 19: Integer Predicates

ieq (Rs1, Rs2, Rd)	Rs1 == Rs2 ? Rd := True : Rd := False
ineq (Rs1, Rs2, Rd)	Rs1 == Rs2 ? Rd := False : Rd := True
ilt (Rs1, Rs2, Rd)	Rs1 < Rs2 ? Rd := True : Rd := False
ile (Rs1, Rs2, Rd)	Rs1 <= Rs2 ? Rd := True : Rd := False
igt (Rs1, Rs2, Rd)	Rs1 > Rs2 ? Rd := True : Rd := False
ige (Rs1, Rs2, Rd)	Rs1 >= Rs2 ? Rd := True : Rd := False

A.4 Integer Arithmetic and Bit Operations

Integer operations take tagged integers as arguments and produce tagged integers as results. Non-integer operands will cause an exception. The integer operations are shown in Table 20.

A.5 Control Transfer

For the control transfer instructions, the destination of transfer is specified by a symbolic identifier `Label` to denote a program point.

Table 20: Integer Operations

iadd (Rs1, Rs2, Rd)	Rd := Rs1 + Rs2
isub (Rs1, Rs2, Rd)	Rd := Rs1 - Rs2
imul (Rs1, Rs2, Rd)	Rd := Rs1 * Rs2
idiv (Rs1, Rs2, Rd)	Rd := Rs1 / Rs2
imod (Rs1, Rs2, Rd)	Rd := Rs1 MOD Rs2
ineg (Rs, Rd)	Rd := - Rs1
iand (Rs1, Rs2, Rd)	Rd := Rs1 AND Rs2
ior (Rs1, Rs2, Rd)	Rd := Rs1 OR Rs2
ixor (Rs1, Rs2, Rd)	Rd := Rs1 XOR Rs2
inot (Rs, Rd)	Rd := NOT Rs

`jump (Label)`

Branch unconditionally to the destination `Label`.

`br (z, Rs, Label)`

If `Rs` is zero, then branch to the destination `Label`. If `Rs` is not zero, the next instruction is executed. If `Rs` is not an integer, an exception is raised.

`br (nz, Rs, Label)`

If `Rs` is not equal to zero, then branch to the destination `Label`. If `Rs` is equal to zero, the next instruction is executed. If `Rs` is not an integer, an exception is raised.

`br (p, Rs, Label)`

If `Rs` is greater than zero, then branch to the destination `Label`. If `Rs` is not greater than zero, the next instruction is executed. If `Rs` is not an integer, an exception is raised.

`br (n, Rs, Label)`

If `Rs` is less than zero, then branch to the destination `Label`. If `Rs` is not less than zero, the next instruction is executed. If `Rs` is not an integer, an exception is raised.

A.6 Unification Instructions

`unify (Rs1, Rs2)`

Perform active unification on the data items passed in the source registers, `Rs1` and `Rs2`. Instruction `unify` will return to the caller only if the active unification was successful. A failure in unification indicates a failed computation and the system is halted.

`punify (Rs1, Rs2, Rd)`

Perform passive unification on the structures in the source registers, and place the status code of the unification into the destination register.

The tagged integer `PUNIFY_SUCCESS` is returned in `Rd` if the passive unification succeeded. If the passive unification fails due to mismatch, then tagged integer `PUNIFY_FAILURE` is returned in `Rd`. In case of uninstantiated variables that cause suspension, the offending variable(s) is(are) pushed on the suspension stack and tagged integer `PUNIFY_SUSPEND` is returned in `Rd`.

A.7 Process Management

`enqueue (Rs)`

`Rs` must be a pointer to a vector structure that represents a goal record. It is not verified that the vector actually represents a goal record. The goal record is put onto the run queue. After the `enqueue` instruction has executed, the value of the source register is to be treated as undefined; under no circumstances may the user code manipulate a goal record after it is enqueued.

`proceed`

The `proceed` instruction terminates the current process.

`push (Rs)`

Push the contents of register `Rs` onto the suspension stack. The implementation of the stack is left undefined in this document.

`execute (Proc, N)`

Transfers control unconditionally to the entry point of a procedure `Proc` which has arity `N`. The suspension stack is cleared before the first instruction of the target procedure is executed.

`suspend (Proc, N)`

Checks the suspension stack. If the stack is not empty, then suspend the procedure `Proc` of arity `N`, on all variables on the suspension stack. If the suspension stack is empty, the computation is a failure, and the system is halted.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1985.

[2] M. Carlsson. *SICStus Prolog User's Manual.* PO Box 1263, S-16313 Spanga, Sweden, February 1988.

[3] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, Royal Institute of Technology, SICS Dissertation Series 02, March 1990.

[4] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[5] J. A. Crammond. *Implementation of Committed-Choice Logic Languages on Shared-Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Endinburgh, May 1988.

[6] J. A. Crammond. personal communication, May 1992.

[7] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, August 1992.

[8] S. Debray, S. Kannan, and M. Paithane. Weighted Decision Trees. In *Joint International Conference and Symposium on Logic Programming.* Washington D.C., MIT Press, November 1992. submitted for publication.

[9] T. P. Dobry. *A High Performance Architecture for Prolog.* Kluwer Academic Publishers, Norwell MA, 1988.

[10] S. Duvvuru, R. Sundararajan, E. Tick, A. V. S. Sastry, L. Hansen, and X. Zhong. A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs. In *International Workshop on Memory Management*, St. Malo, September 1992. ACM Press.

[11] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. Cleveland, MIT Press, October 1989.

[12] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[13] I. Foster and W. Winsborough. Copy Avoidance through Compile-Time Analysis and Local Reuse. In *International Symposium on Logic Programming*, pages 455–469. San Diego, MIT Press, November 1991.

[14] D. Gudeman, K. De Bosschere, and S. K. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*. Washington D.C., MIT Press, November 1992. submitted for publication.

[15] A. Harsat and R. Ginosar. CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog. In *International Conference on Fifth Generation Computer Systems*, pages 962–969, Tokyo, November 1988. ICOT.

[16] J. L. Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-33(11):1221–1246, December 1984.

[17] B. Holmer *et al.* Fast Prolog with an Extended General Purpose Architecture. In *International Symposium on Computer Architecture*, pages 282–291, Seattle, June 1990. IEEE Computer Society.

[18] Intel Corp. *Intel 386 DX Microprocessor Programmer's Reference Manual.* Mt. Prospect IL, 1990.

[19] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society, August 1987.

[20] S. Kliger and E. Y. Shapiro. A Decision Tree Compilation Algorithm for FCP(—,:,?). In *International Conference and Symposium on Logic Programming*, pages 1315–1336. University of Washington, MIT Press, August 1988.

[21] M. Korsloot and E. Tick. A Determinacy Testing Algorithm for Nondeterminate Flat Concurrent Logic Programming Languages. In *International Conference on Logic Programming*, pages 457–471. Paris, MIT Press, June 1991.

[22] P. Kursawe. How To Invent A Prolog Machine. In *International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 134–148. Imperial College, Springer-Verlag, July 1986.

[23] B. Massey and E. Tick. Automatic Mode Analysis for Concurrent Logic Programs: Implementation and Evaluation. Technical Report CIS-TR-92-09, University of Oregon, Department of Computer Science, November 1992.

[24] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1989.

[25] Quintus Computer Systems Inc. *Quintus Prolog User's Guide and Reference Manual—Version 6.* 2100 Geng Road, Palo Alto CA 94303, April 1986.

[26] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.

[27] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[28] M. Sato *et al.* KL1 Execution Model for PIM Cluster with Shared Memory. In *International Conference on Logic Programming*, pages 338–355. University of Melbourne, MIT Press, May 1987.

[29] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[30] P. A. Steenkiste. The Implementation of Tags and Run-Time Tag Checking. In P. Lee, editor, *Topics in Advaced Language Implementation*, pages 3–24. MIT Press, Cambridge MA, 1991.

[31] A. Taylor. LIPS on a MIPS: Results From a Prolog Compiler for a RISC. In *International Conference on Logic Programming*, pages 174–185. Jerusalem, MIT Press, June 1990.

[32] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell MA, 1987.

[33] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA, 1991.

[34] E. Tick. An Optimizing Compiler for Committed-Choice Logic Programs, 1992. in progress.

[35] E. Tick and J. A. Crammond. Comparison of Two Shared-Memory Emulators for Flat Committed-Choice Logic Programs. In *International Conference on Parallel Processing*, volume 2, pages 236–242, Penn State, August 1990.

[36] K. Ueda and T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, 33(6):494–500, December 1990.

[37] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.

[38] P. L. Van Roy. *Can Logic Programming Exectue as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, EECS, 1991. Also available as Technical Report UCB/CSD 90/600.

[39] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[40] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A. V. S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *International Conference on Fifth Generation Computer Systems*, pages 809–816, Tokyo, June 1992. ICOT.