Super Monaco Brothers: The Sequel

A Second Runtime System

J. S. Larson B. C. Massey E. Tick
Department of Computer and Information Science
University of Oregon
{jim,bart,tick}@cs.uoregon.edu

Abstract

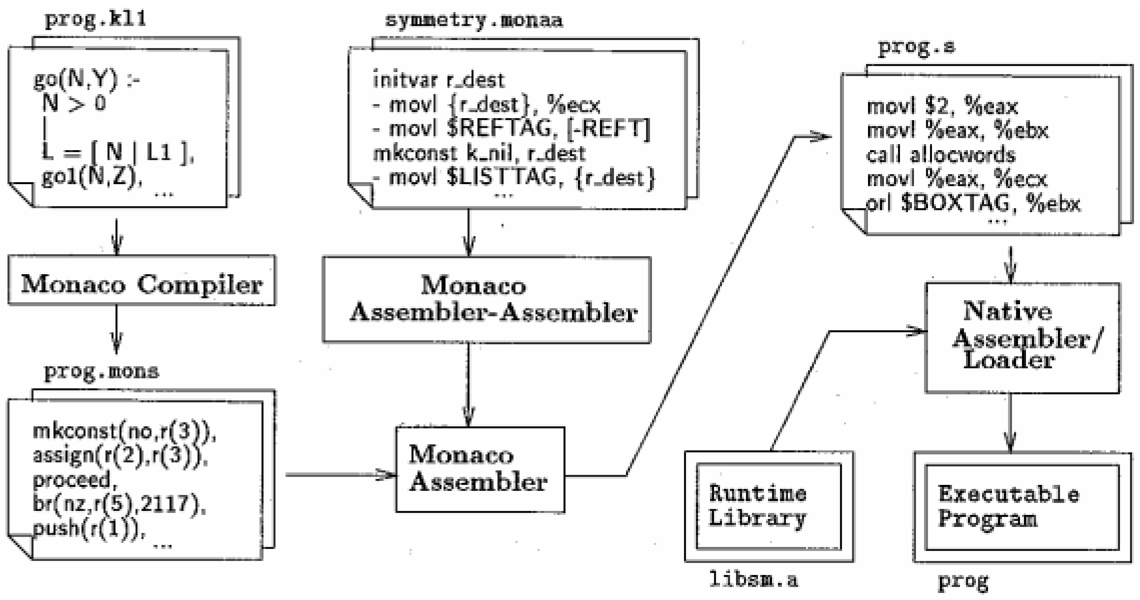
"Super Monaco" is the successor to Monaco, a shared-memory multiprocessor implementation of Flat Guarded Horn Clauses (FGHC). While the system retains the older Monaco compiler and intermediate abstract machine, the intermediate code translator, and the runtime system have been completely replaced, incorporating both the best features of the old architecture and a number of new features intended to improve robustness, flexibility, maintainability, and performance. The compiler, written in KL1, takes high-level FGHC programs and produces intermediate code for the Monaco abstract machine. An "assembler-assembler" converts a host machine description into a KL1 program which translates Monaco intermediate code into target assembly code. There are currently two intermediate code translators: one for SGI MIPS-based hosts, and another for Sequent Symmetry 80386-based multiprocessors. The runtime system, written in C, improves upon its predecessor with better memory utilization and garbage collection, and includes new features such as an efficient termination scheme and a novel variable binding and hooking mechanism. The result of this organization is a portable system (machine description files are about 500 lines, and the runtime system has about 300 lines of machine-dependent C code) which is robust and extensible. This paper describes the design choices made in building the system and the interfaces between the components.

1 Overview

Monaco is a high-performance parallel implementation of a subset of the KL1 [13] dialect of Flat Guarded Horn Clauses (FGHC) [24] for shared-memory multiprocessors. "Super Monaco" is a second-generation implementation of this system, consisting of an evolved intermediate instruction set, a new assembler-generator, and a new runtime system. It incorporates the lessons learned in the first design [21], improves upon its predecessor with better memory utilization (via a 2-bit tag scheme and the use of 32-bit words, as discussed in Section 5) and garbage collection, and includes a number of new features: 1) Termination detection through conservative goal counting. 2) A new mechanism for hooking suspended goals to variables. 3) A specialized language for implementing intermediate code translators. 4) A clean and efficient calling interface between the runtime system and compiled code.

We have found that our changes to Monaco have increased the robustness, portability, and maintainability of the system, while increasing the performance in many cases. The system now has less than 1,000 lines of machine-dependent code, completely encapsulated behind generic interfaces. The new assembler-assembler makes native code generation simple and declarative, while supporting the use of UNIX standard debugging and profiling tools. A conservative goal-counting algorithm implements distributed termination detection. The intermediate code is evolving toward a more abstract machine model, and thus toward more complex instructions. A new data layout makes for more compact use of memory, in conjunction with a novel hooking scheme, which maintains references to suspended goals using a hash table indexed by variable address.

This paper discusses the design choices made in this second-generation system, and its preliminary implementation and performance. Section 2 reviews the Monaco compiler. Section 3 introduces the new assembler-assembler. Section 4 discusses our intermediate code. Section 5 defines the new layout used for our data structures. Section 6 introduces the new runtime system and suspension mechanism. Section 7 gives performance numbers and an evaluation of the new design. Section 8 discusses related work in the literature. Section 9 draws some conclusions.



```
go(N,Y) :-
 N > 0
 |
 L = [ N | L1 ],
 go1(N,Z),   ...
```

```
initvar r_dest
- movl {r_dest}, %ecx
- movl $REFTAG, [-REFT]
mkconst k_nil, r_dest
- movl $LISTTAG, {r_dest}
   ...
```

```
movl $2, %eax
movl %eax, %ebx
call allocwords
movl %eax, %ecx
orl $BOXTAG, %ebx
   ...
```

```
mkconst(no,r(3)),
assign(r(2),r(3)),
proceed,
br(nz,r(5),2117),
push(r(1)),   ...
```

Figure 1: Overview of the Super Monaco System

2 The Monaco Compiler

The Monaco compiler translates programs written in a subset of KL1 [13] to Monaco intermediate code. The compiler is substantially unchanged from the earlier-reported work of [22], but is reviewed here for completeness. The compiler consists of about 2500 lines of "front-end" KL1 code which translates source programs to an intermediate form with explicit decision graphs [14], and about 4500 lines of "back-end" KL1 code which compiles this intermediate form. The process by which a Monaco source language program is compiled is described in Figure 1. A machine description written in a special language is translated into a template based translator. The KL1-subset source program is compiled to an intermediate "assembly" form, which is then translated into native assembly code.

Front-end compilation proceeds in two passes. The first pass translates the source program into a "flattened" form, in which all head structures have been replaced by guard tests, and all variables have been consistently renamed between the clauses of each procedure. The clause heads of the flattened program are then processed to create decision graphs for each procedure, using a variant of Kliger's decision graph algorithm [14].

The back-end first generates code by traversing the intermediate form in a standard fashion, consuming an arbitrary number of pseudo-registers. This code is then passed through an optimizer which builds basic blocks, and performs memory allocation coalescence, constant subexpression elimination, register allocation, and spilling. The next pass shortens jump chains, removes statically decidable branches, and removes dead basic blocks. The basic blocks are then flattened into a linear structure, and a peephole optimizer traverses the resulting code cleaning up some remaining common code-generation inefficiencies.

The number of registers consumed in the target program is limited by a compiler parameter (so that the registers in the intermediate language can be mapped onto general-purpose machine registers of the native-code target) but is otherwise machine-independent. This scheme leads to good portability, while also allowing some experimentation, such as artificially restricting register usage on an architecture to measure performance impacts, or implementing "extra registers" on an architecture using memory locations.

The intermediate code design was originally targeted toward MIPS-based microprocessors, and some vestiges of this decision remain in the compiler. For example, the assumption of a reasonably large number of general-purpose registers (if fewer than about 16 registers are available, code quality degrades substantially) requires the Sequent Symmetry implementation, with only four general-purpose registers available, to implement all of its registers as an array in memory. The assumption that condition-codes are not available as the result of arithmetic and logical computations also leads to some implementation inefficiency on non-MIPS architectures, because explicit logical temporaries are generated and tested, consuming both extra registers and extra instructions.

```
car r_list r_dest
- movl {r_list}, %eax
- movl [-LISTTAG](%eax), %eax
- movl %eax, {r_dest}
incr r_src r_dest
- movl {r_src}, %eax
- addl $[1<<INTSHIFT], %eax
- movl %eax, {r_dest}
sref r_struct n_off r_dest
- movl {r_struct}, %eax
- movl [4*{n_off}-BOXTAG](%eax), %eax
- movl %eax, {r_dest}
ssize r_struct r_dest
- movl {r_struct}, %eax
- movl [-BOXTAG](%eax), %eax
- shrl $[16-INTSHIFT], %eax
- subl $[2<<INTSHIFT], %eax
- movl %eax, {r_dest}
```

(a) Symmetry (i386) Templates

```
car r_list r_dest
- lw {r_dest}, +(-LISTTAG)({r_list})
incr r_src r_dest
- addi {r_dest}, {r_src}, +(1<<INTSHIFT)
sref r_struct n_off r_dest
- lw {r_dest}, +(4*{n_off}-\
     BOXTAG)({r_struct})
ssize r_struct r_dest
- lw {r_dest}, +(-BOXTAG)({r_struct})
- srl {r_dest}, +(16-INTSHIFT)
- sub {r_dest}, {r_dest}, +(2<<INTSHIFT)
```

(b) MIPS Templates

Figure 2: Code Templates for monaa

3 The Monaco Assembler-Assembler

The Monaco intermediate code is referred to, for historical reasons, as "Monaco assembly language." The translator from Monaco intermediate code to target assembly language is thus called mona, the "Monaco assembler." This program has existed in several incarnations: 1) A simple KL1 program was written to translate Monaco intermediate code into 386 assembly code for the Sequent Symmetry. This program suffered somewhat from speed problems, but its main defect was that a succession of inexperienced KL1 programmers found it difficult to understand and maintain. 2) A table-driven C program was written, which could generate either Symmetry or MIPS assembly language. This program was faster than its predecessor, but proved equally difficult to understand and maintain. 3) A machine description language, known as monaa ("Monaco assembler-assembler") was designed. A monaa machine description is automatically translated into KL1 code, and combined with target-independent KL1 code to produce a mona translator for a particular target architecture.

The monaa translator consists of about four hundred lines of awk code, together with a small Bourne shell driver and some m4 macro definitions. The overall structure of the monaa language is that of a simple template expander — no native-code peepholing or other optimizations are currently done, although it is possible that this will change in the future. For each mona instruction, one or more non-overlapping parameterized templates are given, together with machine code produced in response to the match. Type information is attached to both the formal and actual parameters to guide matching and expansion. In addition to instruction templates, the monaa description provides information about register names and calling conventions, as well as some standard templates for procedure prologues and epilogues, debugging information, and the like. The generated native assembly code follows the C calling conventions for linking with the runtime system, and allows for profiling and symbolic debugging of Monaco assembly code using standard UNIX tools. The monaa description for the Sequent Symmetry is about 700 lines of monaa code, expanding to about 1300 lines of KL1. The machine-independent KL1 code for mona comprises about 3400 lines, including symbol-table management and basic housekeeping functionality.

Some of the monaa templates used for current targets are given in Figure 2. Note that the templates in (a) describing the i386 implementation of the Monaco instructions are somewhat larger than those describing the MIPS implementation in (b). This is due in small degree to the two-address nature of i386 instructions (as opposed to MIPS three-address instructions), but largely to the fact that the i386 Monaco registers are actually implemented using memory locations. The small number of general-purpose registers available on the i386 forced this implementation; unfortunately, the Monaco registers thus must be copied to and from real registers in each instruction.

The use of monaa has proved to have several advantages: 1) The specialized machine description language is reasonably easy for non-KL1-literate programmers to use and understand. The bulk of the MIPS machine description was written and debugged in about a week, by an undergraduate with no KL1 experience [2]; the entire MIPS port occupied three people for about a month. 2) The reliance on standard UNIX utilities such as awk, the Bourne shell, sed, and m4 simplifies maintenance of the monaa translator itself. 3) The isolation of machine dependencies facilitates future ports to new architectures. 4) The production of KL1 code makes bootstrap and integrated versions of the assembler straightforward. 5) The ease of modifications to the template has sped up the design and testing cycle dramatically.

```
deref(r(0),r(3))
isbound(r(3),r(4))
br(nz,r(4),14)
push(r(0))
label(13)
suspend(append/3)
label(14)
islist(r(3),r(4))
br(z,r(4),17)
alloc(4,r(7))
initvarref(r(7),1,r(4))
car(r(3),r(6))
initlistref(r(7),2,r(6),r(4),r(5))
assign(r(2),r(5))
cdr(r(3),r(5))
move(r(5),r(0))
move(r(4),r(2))
execute(append/3)
label(17)
isnil(r(3),r(4))
br(z,r(4),13)
unify(r(2),r(1))
proceed
```

Figure 3: Monaco Intermediate Code for **append/3**

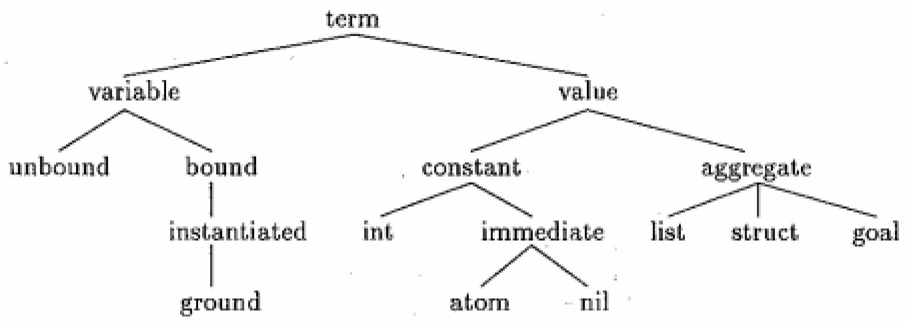


Figure 4: Monaco Object Taxonomy

4 Monaco Intermediate Code

The Monaco instruction set presents an abstract machine which is at an intermediate level between the semantics of a concurrent logic program and the semantics of native machine code. The abstract machine consists of a number of independent processes which execute a sequence of procedures and update a shared memory area. Each process has a set of abstract general-purpose registers which are used as operands for Monaco instructions and for passing procedure arguments. Control flow within a procedure is sequential with conditional branching to code labels. Figure 3 shows the Monaco code produced by the compiler for **append/3**.

The shared memory area is divided into cells, each of which can contain a Monaco data object, also called a *term*. Terms are either *variables* or *values*. Values are either simple *constants* or *aggregates* of terms. The allowed constants are integers, atoms, or the empty list *nil*. The aggregate values are lists or *structs*, which are vectors of terms. A *ground* value is either a constant or an aggregate made up of ground values, that is, an entire structure which contains no variables. Variables may be *bound* to terms. A variable which is bound to a ground value is a *grounded variable*, and grounded variables are themselves ground values. A variable which is bound to a non-variable term is called an *instantiated variable*. If a variable has been bound to another variable, then the instantiation of either variable will cause the instantiation of the other variable to the same value. A taxonomy illustrating these distinctions is given in Figure 4.

Variables are bound through assignment operations or active unification. There are two unification operations. Passive unification verifies the equality of ground values (in contrast to systems such as JAM Parlog [5], which also verify the equality of terms in which uninstantiated variables are bound together). An attempt to passively unify a term containing uninstantiated variables will result in suspension of the process until those variables become instantiated. Active unification, on the other hand, will bind variables to other variables or to values in order to ensure equality of terms. As is customary in logic programming implementations, no "occurs check" is performed during unification for efficiency reasons.

The Monaco instruction set consists of about sixty operations, and is summarized in Tables 1 and 2. The instructions take constants or registers as their arguments and return their results in registers. There is no explicit access to the shared memory except through operations which access the fields of aggregates.

Each data constructor has a variant which serves to batch up allocation requests into a large block, and then initialize smaller sections of the block. Batching up the frequent allocation requests increased

Data Constructors	
`alloc`(n, R_d)	allocate space on the heap
`initgoalref`(R_b, *Offset*, *Size*, *Proc*, R_d)	initialize a goal record
`initlistref`(R_b, *Offset*, R_{s1}, R_{s1}, R_d)	initialize a list
`initstructref`(R_b, *Offset*, *Size*, R_d)	initialize a vector
`initvarref`(R_b, *Offset*, R_d)	initialize a variable
`mkconst`($\langle const \rangle, R_d$)	create a constant
`mkgoal`(*Size*, *Proc*, R_d)	create a goal record
`mklist`(R_{s1}, R_{s2}, R_d)	create a list
`mkstruct`(*Size*, R_d)	create a struct
`mkptr`(R_d, *Label*)	create a pointer to ground data
`mkunbound`(R_d)	create a variable

Ground Data Constructors	
`const`($\langle const \rangle$)	write a constant
`datalabel`(n)	ground table label
`listptr`(*Label*)	write a list pointer
`structptr`(*Label*)	write a struct pointer
`vectorhdr`(*Arity*)	write a struct header

Data Manipulators	
`car`(R_s, R_d)	get head of list
`cdr`(R_s, R_d)	get tail of list
`decr`(R_s, R_d)	compute $R_s - 1$
`incr`(R_s, R_d)	compute $R_s + 1$
`move`(R_s, R_d)	register-to-register copy
`sref`(R_s, n, R_d)	get struct element
`sset`(R_s, n, R_d)	set struct element
`ssize`(R_s, R_d)	get struct size

Predicates	
`eq`(R_{s1}, R_{s2}, R_d)	equal?
`isbound`(R_s, R_d)	instantiated variable?
`isempty`(R_d)	suspension stack empty?
`isimm`(R_s, R_d)	immediate? (atom or nil)
`isatom`(R_s, R_d)	atom?
`isint`(R_s, R_d)	integer?
`islist`(R_s, R_d)	list? (not nil)
`isnil`(R_s, R_d)	nil?
`isstruct`(R_s, R_d)	struct?
`isunbound`(R_s, R_d)	uninstantiated variable?
`neq`(R_{s1}, R_{s2}, R_d)	not equal?
`ieq`(R_{s1}, R_{s2}, R_d)	integer equal?
`ineq`(R_{s1}, R_{s2}, R_d)	integer not equal?
`ilt`(R_{s1}, R_{s2}, R_d)	integer less than?
`ile`(R_{s1}, R_{s2}, R_d)	integer less or equal?
`igt`(R_{s1}, R_{s2}, R_d)	integer greater than?
`ige`(R_{s1}, R_{s2}, R_d)	integer greater or equal?

Table 1: Monaco Instruction Set

Arithmetic and Bit Operations	
`iadd`(R_{s1}, R_{s2}, R_d)	integer add
`isub`(R_{s1}, R_{s2}, R_d)	integer subtract
`imul`(R_{s1}, R_{s2}, R_d)	integer multiply
`idiv`(R_{s1}, R_{s2}, R_d)	integer divide
`imod`(R_{s1}, R_{s2}, R_d)	integer modulus
`iand`(R_{s1}, R_{s2}, R_d)	bitwise and
`ior`(R_{s1}, R_{s2}, R_d)	bitwise or
`ixor`(R_{s1}, R_{s2}, R_d)	bitwise exclusive-or
`ineg`(R_s, R_d)	integer negation
`inot`(R_s, R_d)	bitwise complement

Control	
`br(a,`*Label*`)`	branch always
`br(n,`R_s`,`*Label*`)`	branch on negative
`br(p,`R_s`,`*Label*`)`	branch on positive
`br(z,`R_s`,`*Label*`)`	branch on zero
`br(nz,`R_s`,`*Label*`)`	branch on not zero
`label`(n)	code label

Unification and Process Management	
`assign`(R_{s1}, R_{s2})	bind a variable
`punify`(R_{s1}, R_{s2}, R_d)	passive unification
`unify`(R_{s1}, R_{s2})	active unification
`enqueue`(R_s)	enqueue a goal
`execute(`*Proc/n*`)`	procedure call
`proc(`*Proc/Arity*`)`	marks the beginning of a procedure
`proceed()`	terminate current thread
`push`(R_s)	add to suspension stack
`suspend(`*Proc/n*`)`	suspend a procedure

Table 2: Monaco Instruction Set (cont.)

performance on standard benchmarks, as discussed below in Section 7. In addition, aggregates which are fully ground at compile time are statically allocated in the text segment of the assembled code. This decreases execution and compilation times.

The instruction set is modeled after a reduced instruction set (RISC) architecture, on the theory that such small instructions may be easily and efficiently translated to native RISC instructions using a simple assembler. This is the case for the MIPS port, where many Monaco instructions translate to single MIPS instructions, as shown in Figure 2. However, the Monaco instruction set has been evolving toward more complex instructions, as frequent idioms are identified and coalesced. There are several reasons for this trend: 1) Intermediate instructions at too low a level violate abstraction barriers between the intermediate code and the machine-level data layout and runtime system data structures. 2) As the amount of work per instruction gets larger, more machine-specific optimizations can be made in the `monaa` code templates. 3) There is no reason to equalize the amount of work done per instruction or to standardize instruction formats, as there is with RISC architectures. 4) If the native target is not a good match for the Monaco instruction set, a simple template-expanding assembler will produce much better native code for a more complex instruction than for a sequence of simple instructions. (This is in contrast to systems such as [11], a sophisticated multi-level translation scheme which produces good code by intelligent generation of very simple intermediate instructions.)

5 The Runtime Data Layout

The previous memory layout [21, 6] had three tag bits on each word, and words were laid out on eight-byte boundaries in memory. This prodigious use of memory was not merely a concession to the three tag bits. The unification scheme required each object to be lockable. As a consequence of this requirement, some of the "extra" 32 bits of each word were used as a lock. While this led to a fine granularity for locking, it doubled the system's memory consumption.

All objects are now represented as 32-bit words of memory aligned on four-byte address boundaries. This alignment restriction allows the low-order two bits of pointers to be used as tag bits, without loss of pointer range. The four tagged types are *immediates, list* pointers, *box* pointers, and *reference* pointers.

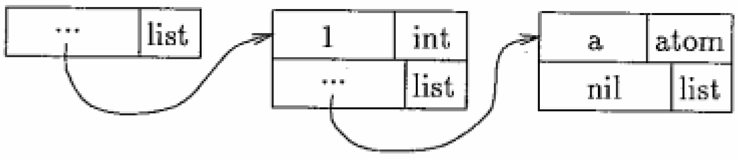


Figure 5: Representation of the List `[1,a]`

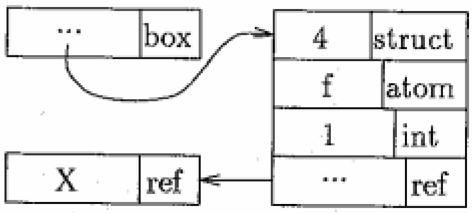


Figure 6: Representation of the Structure `f(1,X)`

Immediates are further subdivided into *integers, atoms,* and box *headers.* Integers have the distinction of being tagged with zero bits, allowing some optimizations to be made in arithmetic code generation. On most architectures, the pointer types suffer no inefficiencies from tagging, since negative offset addressing may be used to cancel the added tag.

List pointers point to the first of two consecutive words in memory, the head and the tail of the list, respectively. The *nil* list is represented as a list-tagged null pointer. Box pointers point to an array of n consecutive words in memory, the first of which is a box header word which encodes the size of the box and the type of its contents. Boxes are used to implement structs, goal records, and strings, as well as some objects specific to the runtime system such as suspension slips. Figures 5 and 6 illustrate the layout of some typical objects.

There is only one mutable object type — the *unbound variable,* represented as a null pointer with a reference pointer tag. When a variable is bound, its value is changed to the binding value. When a variable is bound to another variable, one becomes a reference pointer to the other. Successive bindings of variables create trees of reference pointers which terminate in a root, which is either an unbound variable or some non-variable term. The special Monaco instruction `deref` must thus be applied to all input arguments of a procedure before they are examined. This operation chases down a chain of references to its root, and returns the root value or a reference to the unbound root variable. Thus, a conservative estimate of whether the variable is bound can be made quickly. In practice, this is only a performance issue, not a correctness issue — the process may try to suspend on a recently instantiated variable, in which case the runtime system will detect its instantiation and resume execution of the process.

In the previous implementation of Monaco, one of the tag types was a *hook pointer,* which was semantically equivalent to an unbound variable, but pointed to the set of goal records suspended on that variable. All of the code which dealt with unbound variables also had to test for hook pointers and handle them separately. However, profiling revealed that suspension is a relatively rare event — most variables are never hooked. Therefore the new data layout keeps the association between unbound variables and suspended goal records "off-line," as described in Section 6.4. This new organization seems promising; contention for buckets is indeed rare, and we were able to simplify some critical code sections in unification. However, more evaluation needs to be done.

6 The Runtime System

The runtime system is responsible for memory management, scheduling, unification, and the multiprocessor synchronization involved in assignment and suspension. It consists of about 2000 lines of machine-independent C code, and about 300 lines of machine-dependent C for a particular platform. It has been ported to the Sequent Symmetry and MIPS-based SGI machines.

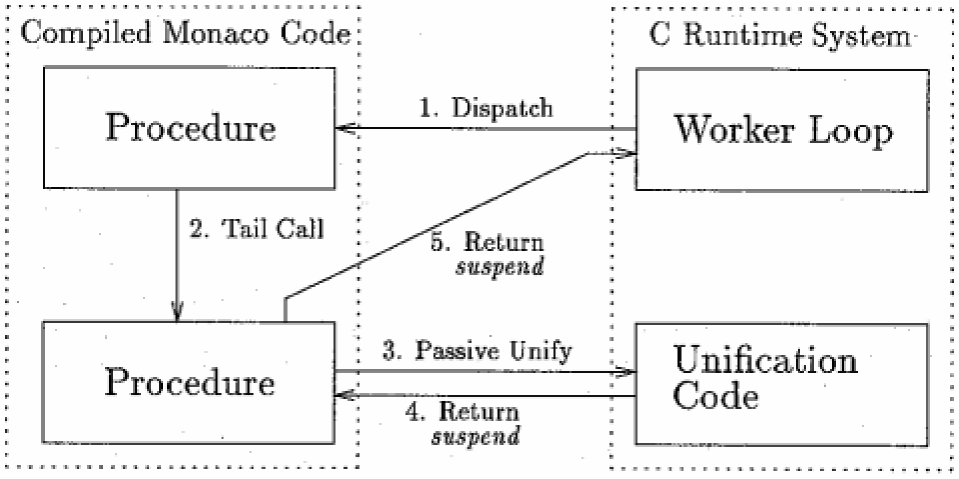


Figure 7: Sample Control Flow in the Monaco System

6.1 Portability

The old Monaco used libraries provided by the host operating system [15] to implement parallel lightweight threads and memory management. We chose to use a more operating system independent model. We create many UNIX processes executing in parallel and communicating through machine-specific synchronization instructions in shared memory, using the `fork` and `mmap` system calls.

The machine-independent portion of the runtime system requires a small set of synchronization primitives from its machine-dependent part. These are: 1) An atomic exchange operation, 2) Atomic increment and decrement, 3) Simple spin locks, 4) Barrier synchronization. These may be operations provided by the architecture, or they may be synthesized from more primitive mechanisms. For the Symmetry port, atomic increment, decrement, and exchange are provided by the instruction set, while locks and barriers are synthesized using the atomic exchange mechanism. The only assumption made in the machine-independent code about the shared memory consistency model is that writes are globally reliable.

The result is a framework that is highly portable, since it does not rely on any particular UNIX implementation's libraries for thread and memory management. The UNIX kernel does the scheduling of the processes on the available processors. Unfortunately, this lead to some drawbacks. UNIX tools designed to interact with implementation-dependent facilities are unavailable. The shared memory must be managed explicitly, since we are not provided with a shared-memory equivalent of `malloc()`. Consequently every runtime system data structure which must be visible to all worker processes must have global scope in the C module system, hindering code modularity. Lastly, the UNIX kernel is not informed of our synchronization operations and may decide to, for instance, preempt a process which is holding a spin lock in order to schedule a process which is waiting for that lock [1]. However, our performance does not currently seem to be impacted by this sort of contention.

The runtime system's interface with the compiled code is small and regular. Implicitly, both components understand the data layout specified in Section 5. The compiled code can only allocate and initialize values; it cannot mutate any values. Thus, all assignment and unification is done in the runtime system. However, there is a runtime data structure which is visible and mutable by both components. This per-worker structure consists of a goal record pointer, used to pass goal records during startup and suspension, a suspension stack, and the limits of a local heap of memory for allocation. This shared structure allows various operations, such as memory allocation and suspension stack management, to be implemented in the compiled code rather than through a function call to the runtime system.

6.2 Scheduling and Calling Interface

The Monaco abstract machine produces many thousands of processes during a typical computation, requiring a level of fine-grained process management inappropriate for implementation via UNIX kernel processes. So, like most concurrent language implementations, we treat UNIX processes (worker processes) as a set of virtual CPUs, on which we schedule Monaco processes in the runtime system.

An invocation of a Monaco process is represented as a goal record, recording simply a procedure name and arguments. A ready set of goal records is maintained by the runtime system. Each worker process starts in a central work loop inside the runtime system. This loop executes until some global termination flag is set, or until there is no more work to do. The worker takes a goal record out of the ready set, loads its arguments into registers, and calls its entry point. The worker then executes a compiled procedure, including sequences of tail calls, until the compiled code decides to terminate, suspend, or fail. These three operations are implemented by a return to the control work loop in the runtime system with a status code as the return value. In addition, the intermediate code instructions for enqueueing, assignment, and unification are implemented as procedure calls from the compiled code into the runtime system. Such calls return back to the compiled code when done, possibly with a status code as a return value. Control flow during a typical execution is illustrated in Figure 7. The runtime system invokes a Monaco procedure via a goal record (1), which tail-calls another procedure (2). This procedure attempts a passive unification via a call into the runtime system (3), which returns a constant *suspend* as an indication that the caller should suspend (4). The caller then suspends by returning the constant *suspend* to the runtime system (5).

The high contention experienced when the ready set is implemented as a shared, locked global object leads to the necessity of some form of distributed ready set implementation. In our scheme, each worker has a fixed-size local ready stack, corresponding to an efficient depth-first search of an execution subtree [17]. If the local stack overflows, local work is moved to a global ready stack. If workers are idle while local work is available, a goal is given to each idle worker, and the remaining local work is moved to the global ready stack. This policy is designed to work well both during normal execution, when many goals are available, and during the initial and final execution phases, when there is little work left to do.

6.3 Termination

Execution of a Monaco program begins when goal records for the calls in the query are inserted into the ready set, and ends when there are no more runnable goals. At this point the computation has either terminated successfully, failed, or deadlocked — the difference can be easily determined in a post-mortem phase which looks for a global failure flag and suspended goals. A serious difficulty for a parallel implementation is efficiently deciding when termination should occur.

Many approaches to termination detection are susceptible to race conditions. The previous implementation maintained a monitor process which examined a status word maintained by each worker process, terminating the computation when it recognized that each work had maintained an idle state for some time. A locking scheme was used to avoid races by synchronizing the workers with the monitor, which hurt worker efficiency. Most importantly, the monitor process itself consumed a great deal of CPU time without performing much useful work.

In Super Monaco, we have adopted a different and (to the best of our knowledge) novel approach. We maintain a count of all outstanding goals — those either in the ready set or currently being executed by workers. Termination occurs when this count goes to zero. The count increases when work is placed in the ready set, and decreases when a goal suspends, terminates, or fails. The count is *not* changed by the removal of a goal from the ready set, since the goal makes a transition from the ready state to the executing state. During the transition interval there is a temporary overestimate of the number of goals outstanding between the time the goal suspends, terminates, or fails, and the time the count is decremented. However, this will not cause premature termination, since the overestimate means that the counter must indicate a nonzero number of outstanding goals. Because the count is not incremented until sometime after a parent has decided to spawn a child goal, there is also a temporary underestimation of the goal count during this interval. As long as the count is incremented before the parent exits, this will not cause premature termination either: Since the parent has not yet exited, the count must be nonzero until after the underestimation is corrected. Thus, since mis-estimates of the number of outstanding goals are temporary and will not cause premature termination, our termination technique is both efficient and safe. On the Symmetry, we implemented this goal counting scheme using atomic increment and decrement instructions: architectures with atomic compare-and-swap should also allow reasonably efficient implementation.

6.4 Hooking and Suspension

In order to awaken suspended processes when a variable becomes instantiated, there must be some association between them. As noted in Section 5, old Monaco represented this association explicitly — some unbound variables were represented as pointers to sets of hooks. Figures 8a illustrates the old representation.

However, for our benchmark set, the vast majority of variables were never hooked. For a variety of reasons, the most important being the fact that we wanted to adopt two bit tag values to represent five

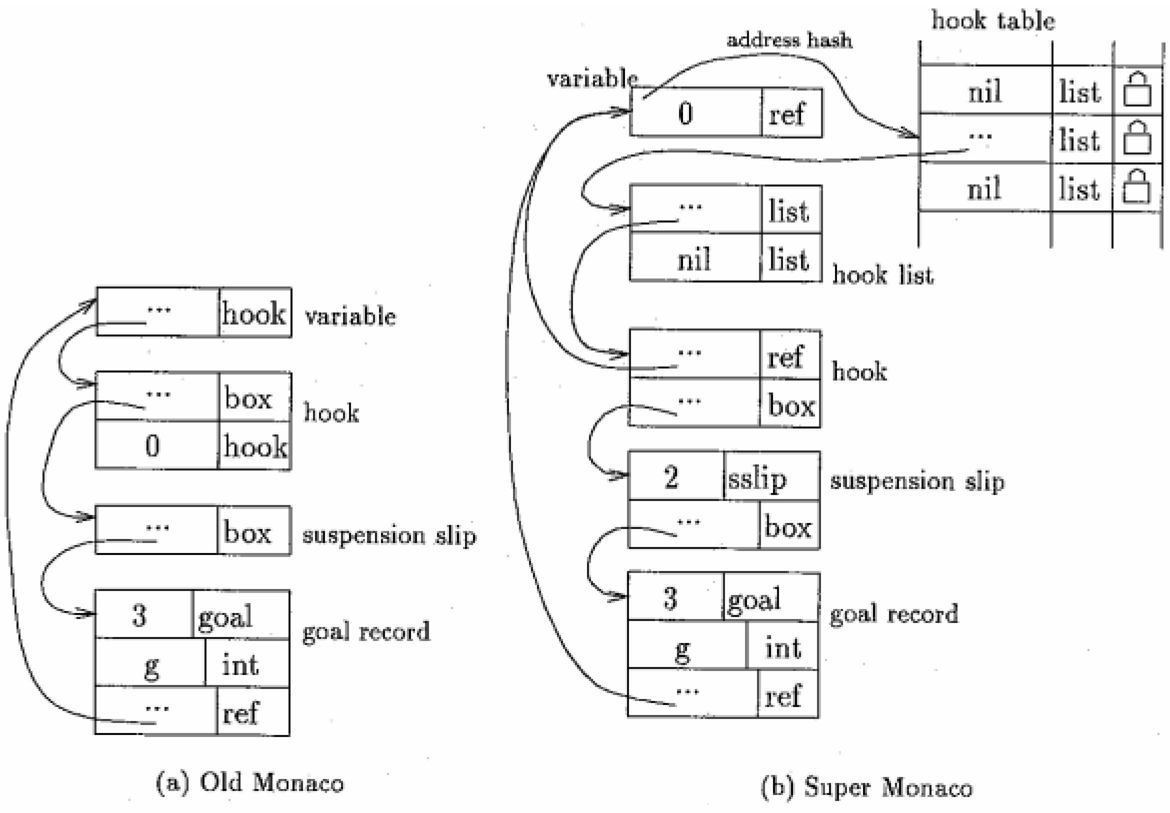


Figure 8: Monaco Hook Structures

types (immediates, lists, box pointers, variable pointers, and reference pointers), we chose to represent variables using a single word. Super Monaco continues to use suspension slips to implement suspension and resumption, as in systems such as JAM Parlog [5] and PDSS [13], except that the association between variables and hooks is reversed. Each hook contains a pointer to the variable it is suspended upon. Hooks are grouped into sets according to a hashing function based upon variable addresses. A global hook table contains a lock for each such set.

Since any operation on an uninstantiated variable necessarily involves the manipulation of the hook table, the locks on the buckets of the hook table may serve as the only synchronization points for assignment and unification. This gives a lower space overhead for the representation of variables on the heap. There will be some hash-related contention for locks which would not occur in a one-lock-per-variable scheme, but since we are dealing with shared-memory machines with a moderate number of processors, the rate of such hash collisions can be made arbitrarily low by increasing the size of the hook table.

To instantiate a variable, its bucket is locked, the unbound cell is bound to its new value, all corresponding hooks are removed from the bucket, and the lock is unlocked. All hooks are then examined. To bind a variable to another variable, both buckets are locked (a canonical order is chosen to prevent deadlock) and the set of hooks of on the second variable are extracted and mutated into hooks on the first variable. These hooks are then placed in the first variable's bucket, and the second variable is mutated into a reference to the first. The result is that future dereferencing operations will return a reference to the new root, or its value when instantiated. Figure 8b illustrates the new representation.

6.5 Memory Management

Memory is allocated in a two-tiered manner. First, there is a global allocator which allocates blocks of memory from the shared heap. Access to the global allocator is sequentialized by a global lock. Second, each worker uses the global allocator to acquire a large chunk of memory for its private use. All memory allocation operations attempt to use this private heap, falling back on the global allocator when the private heap is exhausted. When the global heap is exhausted, execution suspends while a single worker performs a stop-and-copy garbage collection [3] of the entire heap. Garbage collection overheads are acceptably low now, but a parallel garbage collector will be implemented in the near future.

Many runtime system data structures are allocated on the heap and represented as Monaco objects.

Benchmark	System	Processors 1	2	4	8	12	16
hanoi(14)	old	4.364	2.339	1.218	0.772	0.598	0.558
	new	3.240	1.690	0.870	0.450	0.310	0.250
nrev(1000)	old	19.153	11.740	6.136	3.394	2.503	2.155
	new	30.610	15.910	8.300	4.450	3.280	2.490
queen(10)	old	43.305	30.348	12.413	6.153	4.041	3.082
	new	41.620	21.190	10.710	5.410	3.650	2.770
primes(5000)	old	12.841	7.549	3.788	1.984	1.390	1.117
	new	20.730	10.720	5.700	3.150	2.270	1.700
pascal(200)	old	8.973	4.981	2.499	1.294	0.919	0.724
	new	10.590	5.510	2.890	1.540	1.070	0.860

Table 3: Comparison of Execution Times (Seconds)

	Processors 1	2	4	8	12	16
Unification	43.5	43.6	43.5	41.8	38.7	38.4
Scheduling	5.7	6.3	7.5	9.5	13.1	14.7
Suspension	0.0	0.0	0.3	1.0	2.1	1.7
Contention	0.0	0.0	0.0	0.0	0.1	0.3
Runtime Alloc.	0.0	0.0	0.1	0.6	1.3	1.1
Compiled Alloc.	3.1	3.0	2.9	2.6	2.6	2.3
Other Compiled	47.7	47.1	45.8	44.5	42.0	41.5

Table 4: Execution Time Breakdown (by Percentage)

Most sets are currently represented using lists, including the ready set and the hook set in a hook table entry. Strings, which are allocated by the parser, are stored as special boxes. Suspension hooks and suspension slips are stored in list cells and small boxes respectively. Heap allocation of these objects makes it possible to avoid built-in limits on object sizes, and sped development time through the reuse of existing code. However, implementing these facilities using statically-allocated resources, such as a fixed-size ready set and fixed-size suspension stacks, would not only reduce memory-allocation overhead, but would reduce contention by shortening critical sections. Any brittleness due to fixed-size structures in such an implementation could be handled by falling back on dynamic allocation when stacks grow beyond their static limits.

7 Performance and Evaluation

Figure 3 compares Super Monaco performance to that of the previous implementation, as measured by executing the benchmark programs listed in [22] under identical conditions. The benchmarks were executed on a Sequent Symmetry S81 with 16MHz Intel 80386 microprocessors. Measured execution times reflect the user-level CPU time consumed by the longest running processor from the beginning of the computation until termination. All times are the best of several runs (5–10 for old Monaco, 5 for Super Monaco).

In many cases Super Monaco matches or improves on the performance of the previous system, despite the fact that it is more robust, and has not yet incorporated all of the optimizations used in the previous system. Tick and Banerjee [22] compared the old Monaco's performance to that of comparable systems available at the time, such as Strand [7], JAM [5], and Panda [17]. Monaco was found to outperform these systems in a uniprocessor configuration by factors ranging from 1.6 to 4.0, and to maintain such ratios for moderate numbers (1–16) of processors. The new implementation of Monaco still maintains competitive performance.

The first implementation of the runtime system gave us measurements [22] which guided Super Monaco. For the benchmarks we studied, most variables were never hooked; assignment operations should therefore be streamlined for this case even at the expense of the suspending case. We confirmed old Monaco measurements indicating that even if memory was allocated from a private per-processor

	Processors				
	2	4	8	12	16
Scheduling	0.0	45.0	79.3	95.1	96.0
Unification	0.0	20.0	9.0	2.4	1.9
Suspension	0.0	0.0	3.8	1.0	0.9
Barriers	0.0	35.0	7.9	1.6	1.2
Total Contentions	0	20	266	1991	3806

Table 5: Locking Contentions (by Percentage)

area, and even if the allocation instructions were inlined in the compiled code, it was still important to reduce the number of allocation operations, by doing all allocation requests in a basic block at once. We found that global ready-set operations indeed had high contention. Finally, we discovered that the intermediate code's method of computing boolean tests into registers and then branching on register contents did not map well to non-MIPS architectures, and even on the MIPS had no particular advantage.

Although we have not yet incorporated all of the optimizations used in old Monaco, we can use profiling data to identify the limitations on our performance and attempt to analyze how the known optimizations can improve them. The **mona** assembler lets us generate the information necessary for profiling our compiled code using standard UNIX tools. We analyzed the performance of compiled code and the runtime system using the UNIX **gprof** facilities. Table 4 gives the breakdown of the execution time for various levels of parallelism. We note that the runtime overhead due to unification is extremely high, because it has not been tuned yet.

Several sources of overhead remain:

- The use of a new data layout conflicts with the Monaco compiler's "knowledge" of data representations, thus causing a semantic mismatch in its use of the intermediate code **assign** instruction. Therefore all assignments are done with unification in Super Monaco. We believe that a combination of simple static analysis and runtime system tuning will improve assignment performance.

- The previous implementation experienced a performance increase of from 5% to 17% when using a **switch** intermediate instruction which replaced the MIPS-style condition registers. We expect to achieve this advantage as well, although in a cleaner fashion, by modifying the Monaco instruction set slightly. This should reduce the execution overhead of compiled code.

- The new runtime system fixed a number of correctness bugs that plagued the old system, and introduced new capabilities such as garbage collection. These changes led to some additional overheads.

The scalability of the system to larger numbers of processors is limited by the increasing overhead of scheduling operations and the overhead of shared lock contention. Table 5 shows the sources of lock contention. Almost all collisions are due to scheduling operations, and these contentions are a negligible fraction of the time spent in scheduling. The entries for Unification and Suspension indicate that hash collisions of variable addresses in the hook table are not a significant source of contention.

8 Related Work

Among the first abstract machine designs for committed-choice languages were an implementation of Flat Concurrent Prolog [18] by Houri [12, 19], the Sequential Parlog machine by Gregory *et al.* [8, 9], and the KL1 machine by Kimura [13] at ICOT. A good summary of work on Parlog appears in Gregory's book [8]. The JAM Parlog system [5] is a commonly-used Parlog implementation which compiles Parlog into code for an abstract machine interpreter. The implementation of JAM Parlog features many innovations which are still in current use by both our system and others, including hangers, suspension slips, tail call optimization, and goal queues. In spite of its interpreted nature, JAM Parlog is quite efficient. An outgrowth of work on Flat Parlog implementation, the Strand Abstract Machine [7] was originally designed for distributed execution environments, but also achieved excellent performance on shared-memory parallel machines. Work on shared-memory parallel implementations of the committed-choice language KL1 at ICOT led to the KL1C (KL1 C compiler) system [4], which translates KL1 code into portable C code. While the initial KL1C implementation is uniprocessor based, a distributed-memory multiprocessor version [16] is under development. Other recent work has included the **jc** Janus implementation

[10]. Interesting comparisons of the execution performance of many early committed-choice language implementations can be found in Taylor [20] and Tick [22, 23].

9 Conclusions

Super Monaco has achieved improvements over its predecessor in robustness and capability, and some increases in performance. For the shared-memory hosts we are targeting, we have concluded that: 1) A machine description language, automatic translator generation, and a carefully written runtime system all facilitate porting. 2) Generating native assembly code permits access to most of the environment-related advantages of generating C code (easy profiling, debugging, and linking) with only a little extra effort. 3) Termination detection through conservative goal counting is both efficient and simple. 4) A reasonably simple implementation of a distributed ready set can almost entirely eliminate scheduling contention. 5) Implementing suspension using a hook table provides acceptable performance, and allows simpler implementation of Super Monaco as a whole. More improvements aimed at "fastpathing" the non-suspension cases of unification are desirable. 6) A migration toward more complex intermediate-code instructions and greater abstraction from the underlying implementation than were present in old Monaco is desirable, to improve performance without sacrificing portability.

In the immediate future we plan to attack performance bottlenecks. We need to further reduce the cost of unification by identifying assignments and treating them specially, and "fastpathing" operations on unhooked variables — since suspensions are rare, we can more aggressively trade off the expense of suspension operations for streamlined handling. We also need to coalesce the instruction set further, especially type tests.

Acknowledgements

J. Larson was supported by a grant from the Institute of New Generation Computer Technology (ICOT). B. Massey was supported by a University of Oregon Graduate Fellowship. E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[2] C. Au-Yeung. A RISC Backend for the 2nd Generation Shared-Memory Multiprocessor Monaco System. Undergraduate thesis, University of Oregon, December 1994. In Progress.

[3] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1978.

[4] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 25–39, Madrid, September 1994. Springer-Verlag.

[5] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10(4):385–422, August 1992.

[6] S. Duvvuru. Monaco: A High Performance Implementation of FGHC on Shared-Memory Multiprocessors. Master's thesis, University of Oregon, June 1992. Also available as Technical report CIS-TR-92-16.

[7] 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.

[8] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

[9] S. Gregory, I. Foster, A. Burt, and G. Ringwood. An Abstract Machine for the Implementation of Parlog on Uniprocessors. *New Generation Computing*, 6:389–420, 1989.

[10] 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*, pages 399–413. Washington D.C., MIT Press, November 1992.

[11] R. C. Haygood. Native Code Compilation in SICStus Prolog. In *International Conference on Logic Programming*, Genoa, June 1994. MIT Press.

[12] A. Houri and E. Y. Shapiro. A Sequential Abstract Machine for Flat Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 513–574. MIT Press, Cambridge MA, 1987.

[13] 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 Press, August 1987.

[14] S. Kliger and E. Y. Shapiro. From Decision Trees to Decision Graphs. In *North American Conference on Logic Programming*, pages 97–116. Austin, MIT Press, October 1990.

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

[16] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed Memory Implementation of KLIC. In T. Chikayama and E. Tick, editors, *Proceedings of the ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, Eugene, March 1994. University of Oregon Technical Report CIS-TR-94-04.

[17] 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.

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

[19] W. Silverman, M. Hirsch, A. Houri, and E. Y. Shapiro. The Logix System User Manual, Version 1.21. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 46–77. MIT Press, Cambridge MA, 1987.

[20] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[21] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 266–278. Springer Verlag, June 1993.

[22] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming*, pages 757–773. Budapest, MIT Press, June 1993.

[23] 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.

[24] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA., 1987.