Super Monaco: Its Portable and Efficient Parallel Runtime System

J. S. Larson B. C. Massey E. Tick

University of Oregon, Eugene OR 97403, USA

Abstract. "Super Monaco" is the successor to Monaco, a shared-memory multiprocessor implementation of a flat concurrent logic programming language. While the system retains, by-and-large, the older Monaco compiler and intermediate abstract machine, the intermediate code translator and the runtime system have been completely replaced, incorporating a number of new features intended to improve robustness, flexibility, maintainability, and performance. There are currently two native-code backends for 80x86-based and MIPS-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[1] which is robust, extensible, and has performance competitive with C-based systems. This paper describes the design choices made in building the system and the interfaces between the components.

KEYWORDS: logic programming, parallelism, naive code, runtime systems.

1 Introduction

Monaco is a high-performance parallel implementation of a subset of the KL1, a concurrent logic programming language [15], 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 [17], improves upon its predecessor with better memory utilization (via a 2-bit tag scheme and the use of 32-bit words) 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. The system now has less than 1,000 lines of machine-dependent code, completely

[1] Available by anonymous ftp from `ftp.cs.uoregon.edu:pub/bart/sm.tar.gz`.

encapsulated behind generic interfaces. The new assembler-assembler makes native code generation simple and declarative, while supporting the use of 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 with a hash table indexed by variable address.

This paper discusses the design choices made in this second-generation system, its implementation and performance. Because of space limitations, we cannot review the compiler and assembler here (see Tick *et al.* [19]).

2 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. See Tick *et al.* [19] for examples of Super Monaco intermediate code.

There are two unification operations. Passive unification verifies the equality of ground values (in contrast to systems such as JAM Parlog [4], 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. Variables are bound only through assignment operations or active unification.

The Monaco instruction set consists of about sixty operations. The operations are broadly categorized as: 1) Data constructors for each data type (constant, list, struct, goal record, variable). 2) Data manipulators for accessing the fields of aggregates. 3) Arithmetic operations. 4) Predicates for testing the types of most objects and for arithmetic comparisons. Predicates store the truth value of their result in a register. 5) Conditional branches based on the contents of a register. 6) Interfaces to runtime system operations for assignment, unification, suspension, and scheduling. 7) Instructions for manipulating the suspension stack. 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 performance on standard benchmarks, as discussed below in Section 5. 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 with a simple assembler. However, the Monaco instruction set has been evolving toward more complex instructions, as frequent idioms are identified and coalesced. There a couple of reasons for this trend: 1) Higher-level intermediate instructions better hide the runtime system implementation. 2) As the amount of work per instruction gets larger, more machine-specific optimizations can be made in the `monaa` code templates. (This is in contrast to systems such as [8], a sophisticated multi-level translation scheme which produces good code by intelligent generation of very simple intermediate instructions.)

3 The Runtime Data Layout

The previous memory layout [17] 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. 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 objects specific to the runtime system such as suspension slips.

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 original 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." This new organization seems promising (see Section 4.4); contention for buckets is indeed rare, and we were able to simplify some critical code sections in unification.

4 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.

4.1 Portability

The old Monaco used libraries provided by the host operating system [12] 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-dependent runtime system requires only a few synchronization primitives. These are: 1) An atomic exchange operation, 2) Atomic increment and decrement, 3) Simple spin locks, 4) Barrier synchronization. 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 machine-independent code assumes globally reliable writes. The runtime system's interface with the compiled code is small and regular.

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, but there are tradeoffs. UNIX debuggers are too low level. The shared memory must be managed explicitly; consequently, every runtime system data structure which must be visible to all worker processes be a C global, hindering code modularity. The UNIX scheduler may interact badly with our threads [1], though we have not seen this.

4.2 Scheduling and Calling Interface

The Monaco abstract machine produces many thousands of processes during a typical computation, too many for implementation via UNIX kernel processes. We treat UNIX "worker" processes as a set of virtual CPUs, on which we schedule Monaco processes in the runtime system.

A *goal record* records the procedure name and arguments of a Monaco Process. 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 terminates, suspends, or fails. 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 1. 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).

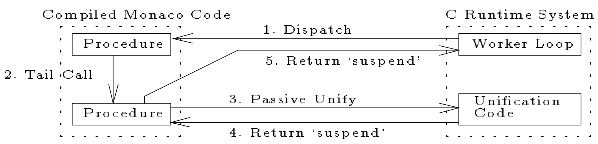


Fig. 1. Sample Control Flow in the Monaco System

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 [14]. 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 to do.

4.3 Termination

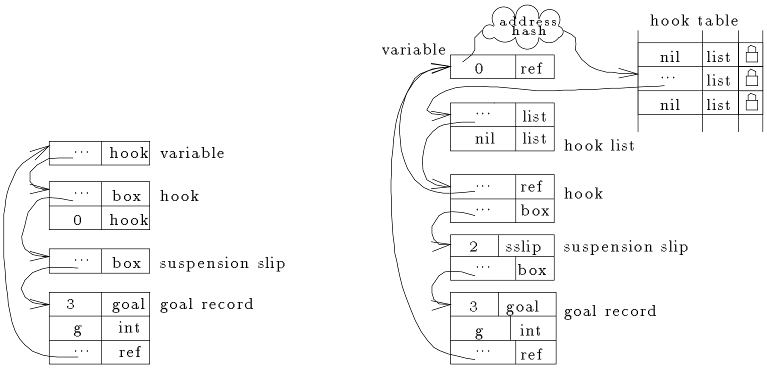
Execution of a Monaco program begins when goal records for the calls in the query are inserted into the ready set, and ends when there are no more runnable goals. At this point the computation has either terminated successfully, failed, or deadlocked — the difference can be easily determined in a post-mortem 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. There is a temporary overestimate of the number of goals outstanding during the transition interval 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 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 with atomic increment and decrement instructions.

4.4 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 3, old Monaco represented this association explicitly: some unbound variables were represented as pointers to sets of hooks. Figures 2a illustrates the old representation.



(a) Old Monaco (b) Super Monaco

Fig. 2. Monaco Hook Structures

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 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 [4] and PDSS [10], 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 2b illustrates the new representation.

To evaluate the performance of our hooking scheme, we replaced it with a more traditional technique. In the latter approach, a list of suspension slips for goals suspended on an unbound variable is maintained in the cell following the variable on the heap. When the variable is bound, the binding process picks up the list directly: the garbage collector will eventually reclaim the extra cell. The traditional implementation requires a locking scheme for variables. We adopt the convention that a locked variable is represented by a reference to itself, i.e., to the location of the locked variable. This representation has an interesting advantage: readers of the variable will spin dereferencing its location until the lock is released, and thus do not have to be modified to be aware of variable locking. The actual lock operation is conveniently implemented with atomic exchange on architectures which have this capability.

Table 1 shows the performance comparison (see Section 5 for benchmark descriptions). In general, there is insignificant performance difference between the two representations (the poor performance of wave needs further investigation). In general, most of the differences are due to the longer typical-case path length of the table-based scheme (20 instructions versus 15), which in turn is an unavoidable consequence of the scheme's more complex nature. Although the two-cell representation is slightly faster, future runtime system optimizations may reverse this advantage.

	hooking performance			**inlining performance**		
benchmark	**two-cell**	**hash table**	**slowdown**	**inlined**	**non-inlined**	**slowdown**
hanoi(14)	2.3	2.4	4%	2.3	2.5	9%
nrev(1000)	11.9	13.1	10%	11.9	16.0	34%
pascal(200)	4.1	4.2	2%	4.1	4.7	14%
primes(5000)	9.3	9.8	5%	9.3	11.3	21%
queen(10)	28.3	30.5	7%	28.3	29.1	3%
cube(6)	38.0	38.9	2%	38.0	38.0	0%
life(20)				20.5	20.9	2%
semigroup	140.7	147.8	5%	140.7	142.8	1%
waltz	26.6	27.0	2%	26.6	27.0	2%
wave(8,8)	7.4	9.2	24%	7.4	7.7	4%

Table 1. Hooking Scheme and Inlining Performance Impacts (Seconds, Symmetry)

4.5 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 of the entire heap. Garbage collection overheads are acceptably low now, but a parallel garbage collector will be implemented in the near future.

The heap holds not only objects created by the compiled code, but also dynamically created runtime system structures. 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. Sets of objects are either represented as statically-limited tables (such as suspension stacks) or as lists (such as hook lists). All sets were first implemented as lists on the heap, avoiding static limits on set sizes, and also speeding development time through reuse of general-purpose code. However, using statically-allocated resources not only reduces memory-acllocation overhead, but also reduces contention by shortening critical sections. If no reasonable limit to set size is known at compile time, such as for the set of ready goals, a hybrid scheme is used where dynamically allocated storage is used to handle the overflow of statically-allocated tables.

4.6 Unification

In early benchmarking, we found that the high frequency of active unification made it a performance bottleneck. We have largely solved this problem through the implementation of "fast paths" through the active unification process. The approach is based on the Monaco compiler's identification of certain active unifications as *assignments* whose left-hand side is likely (but not certain) to be a reference directly to an unbound, unhooked variable, and whose right-hand side is likely to to be a bound value. Assignments comprise the bulk of active unification performed during execution.

The main optimization of assignments is to arrange for inline assembly code to test that the conditions for the assignment are met, and if so, perform the assignment inline. If the assignment is too complex to perform inline, it is passed to a specialized procedure which attempts to optimize some additional common cases. Thus the general active unifier is infrequently executed.

Table 1 shows the performance of the inlined and non-inlined versions (for the two-cell scheme). Differences are substantial in several benchmarks, and in no case do the extra tests degrade performance. For example, `nrev(1000)` performs about 500,000 assignments (and 1000 general unifications). Of the assignments, all but 12 are handled inline, resulting in 34% overall performance improvement.

5 Performance Evaluation

Super Monaco was evaluated on two sets of benchmarks executed on a Sequent Symmetry S81 with 16MHz Intel 80386 microprocessors. The first set, consisting of small, standard programs, is used for comparisons with other systems: KLIC [3] and Monaco. The second set, containing larger programs, is used for runtime system analysis. Table 2 compares Super Monaco, (original) Monaco [18], and

KLIC (uniprocessor version) performance. All times are the best of several runs, using the sum of user- and system-level CPU times. In all cases, Super Monaco improves on the performance of the previous system, despite the fact that it is more robust. Tick and Banerjee [18] compared the old Monaco's performance to that of comparable systems available at the time, such as Strand [5], JAM [4], and Panda [14]. 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 1–16 processors (PEs). The new implementation of Monaco maintains this competitive performance. The uniprocessor performance relative to KLIC is respectable for all benchmarks, and especially good for the larger, more realistic benchmarks, where the geometric mean slowdown is only 20%.

	benchmark	**KLIC**	**Super Monaco**	**KLIC:SM**	**original Monaco**	**Monaco:SM**
small	hanoi(14)	0.6	2.3	0.261	4.4	1.91
	nrev(1000)	5.9	11.9	0.495	19.2	1.61
	pascal(200)	1.7	4.1	0.415	9.0	2.19
	primes(5000)	4.4	9.3	0.474	12.8	1.37
	queen(10)	10.4	28.3	0.368	43.4	1.53
large	cube(6)	15.5	38.0	0.408		
	life(20)	29.6	20.5	1.45		
	semigroup	85.9	140.7	0.610		
	waltz	18.8	26.6	0.709		
	wave(8,8)	11.6	7.4	1.56		

Table 2. Comparison of Uniprocessor Performance (Seconds, Symmetry)

Tick *et al.* [19] present the multiprocessor execution times of Super Monaco. Times were measured for the longest running PE from the beginning of the computation until termination. The geometric mean speedups of the small benchmarks (as defined above) on 16 PEs are 10.3, 10.7, and 11.1 for Super Monaco, old Monaco, and JAM Parlog. However, the geometric mean execution times on 16 PEs are 0.76, 1.2, and 3.0 seconds, respectively. For the large benchmarks, Super Monaco achieves a geometric mean of 10.5 speedup on 16 PEs.

The `mona` assembler facilitates profiling our compiled code with standard UNIX tools. We analyzed the performance of compiled code and the runtime system using the UNIX `prof` facilities. Table 3 gives the breakdown of the execution time for differing numbers of PEs, as an arithmetic mean percentage over the larger benchmarks. The top portion of the table is runtime system overheads. The bottom portion is compiled thread execution. Runtime Alloc. and Compiled Alloc. are memory allocation overheads (not including GC). System scalability to larger numbers of PEs is limited by the increasing overhead of scheduling operations and the overhead of shared lock contention. We believe that almost all lock collisions are due to scheduling operations. The system is not yet balanced, with compiled code running below 40% of total execution time; however, these statistics are influenced a great deal by the benchmark suite.

	1 PE	2 PE	4 PE	8 PE	12 PE	16 PE
Unification	30.6	29.2	27.7	26.6	24.4	21.0
Scheduling	16.9	17.0	16.9	16.6	17.6	16.8
Suspension	4.8	5.1	5.0	5.3	4.1	3.6
Runtime Alloc.	5.1	5.6	6.3	6.1	5.7	4.7
Idling	0.2	1.6	3.4	4.3	6.0	9.8
Contention	0.0	0.0	0.1	1.0	3.2	7.8
Compiled Code	41.2	40.2	39.5	39.1	37.9	35.5
Compiled Alloc.	1.3	1.3	1.1	1.1	1.1	0.9

Table 3. Execution Time Breakdown (by Percentage)

6 Related Work

Among the first abstract machine designs for committed-choice languages were an implementation of Flat Concurrent Prolog [15] by Houri [9, 16], the Sequential Parlog machine by Gregory *et al.* [6], and the KL1 machine by Kimura [10] at ICOT. A good summary of work on Parlog appears in Gregory's book [6]. The JAM Parlog system [4] 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 tail call optimization and goal queues. In spite of a layer of emulation, JAM Parlog is reasonably efficient. An outgrowth of work on Flat Parlog implementation, the Strand Abstract Machine [5] was originally designed for distributed execution environments, but also achieved good performance on shared-memory parallel machines. More recent work includes the ICOT KLIC system [3], which translates KL1 code into portable C code, achieving excellent performance. Uniprocessor and distributed-memory versions [13] have been released. The `jc` Janus system is a similar uniprocessor-based, high-performance implementation [7]. See Chikayama [3] for an in-depth performance comparison among KLIC, `jc`, Aquarius Prolog, and SICStus Prolog.

7 Conclusions

Super Monaco has obsoleted its predecessor in robustness, capability, and execution performance, on the shared-memory hosts we are targeting. The novel contribution of this paper is the development of a real-parallel concurrent logic programming language implementation that achieves speeds competitive with the fastest known uniprocessor implementations, while retaining speedups comparable to the best shared-memory implementations. Other contributions include an efficient termination detection algorithm, a new hooking scheme, an assembler-assembler framework that facilitates portability, and support for native profiling, debugging, and linking. Future work includes exploring optimizations, such as lazy resumption and uses of mode analysis, to further reduce overheads.

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. We thank C. Au-Yeung and N. Badovinac for their help with this research.

References

1. T. Anderson *et al.* Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. on Comp. Sys.*, 10(1):53–79, 1992.
2. C. Au-Yeung. A RISC Backend for the 2^{nd} Generation Shared-Memory Multiprocessor Monaco System. Bachelor's thesis, University of Oregon, December 1994.
3. T. Chikayama *et al.* A Portable and Efficient Implementation of KL1. In *Int. Symp. on Prog. Lang. Impl. and Logic Prog.*, pp. 25–39, 1994. Springer-Verlag.
4. J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10(4):385–422, August 1992.
5. I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conf. on Logic Prog.*, pages 497–512. MIT Press, October 1989.
6. S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.
7. D. Gudeman *et al.* jc: An Efficient and Portable Sequential Implementation of Janus. In *Joint Int. Conf. and Symp. on Logic Prog.*, pp 399–413. MIT Press, 1992.
8. R. C. Haygood. Native Code Compilation in SICStus Prolog. In *International Conference on Logic Programming*, pages 190–204, Genoa, June 1994. MIT Press.
9. A. Houri *et al.* A Sequential Abstract Machine for Flat Concurrent Prolog. In *Concurrent Prolog: Collected Papers*, vol. 2, pp. 513–574. MIT Press, 1987.
10. Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Int. Symp. on Logic Prog.*, pp. 468–477. IEEE Computer Society Press, 1987.
11. S. Kliger and E. Y. Shapiro. From Decision Trees to Decision Graphs. In *North American Conf. on Logic Prog.*, pages 97–116. MIT Press, October 1990.
12. A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1989.
13. K. Rokusawa *et al.* Distributed Memory Implementation of KLIC. *New Generation Computing*, vol. 14, no. 3, 1995.
14. M. Sato *et al.* Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Work. Conf. on Par. Processing*, pp. 305–318. North Holland, 1988.

15. E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.
16. W. Silverman *et al.* The Logix System User Manual, Version 1.21. In *Concurrent Prolog: Collected Papers*, vol. 2, pp. 46–77. MIT Press, 1987.
17. E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *Conf. on Parallel Arch. and Lang. Europe*, LNCS no. 694, pp. 266–278. Springer Verlag, 1993.
18. E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *Int. Conf. on Logic Prog.*, pages 757–773. MIT Press, June 1993.
19. E. Tick *et al.* Experience with the Super Monaco Optimizing Compiler. University of Oregon, Dept. of Computer Science Technical Report CIS-TR-95-07.