Performance Evaluation of the Monaco Compiler and Runtime Kernel

E. Tick and C. Banerjee

CIS-TR-92-21a
April 1993

Abstract

Monaco is a native-code, shared-memory multiprocessor testbed implementation of flat committed-choice languages such as Flat Guarded Horn Clauses. A reduced abstract-machine instruction set facilitates decision-graph code generation and traditional optimizations based on dataflow analysis. We describe the compilation process and present empirical measurements characterizing performance gains enabled by various compilation and runtime kernel construction techniques. Monaco ran 1.6–4 times faster than comparable parallel systems, yet achieved slightly better multiprocessor speedups (10.7 on 16 Symmetry processors). Among the compiler optimizations, common subexpression elimination achieved the largest speedup of 15%.

This report is an extended version of a paper appearing in the *International Conference on Logic Programming*, Budapest, June 1993.

Printing history: TR-92-21, November 1992.

Department of Computer and Information Science
University of Oregon

1 Introduction

High-performance logic programming language implementations require advanced compilation techniques. Specialized hardware may reduce execution times compared to conventional microprocessors, but the *utility* of these optimizations can be low. Compiler optimizations have higher utility because the compiler is usually customized for a particular language family, whereas hardware must be generalized to all languages of practical interest. After initial forays building logic programming machines (e.g., [8, 24, 23]) based on the Warren Abstract Machine (WAM) [2], advanced compilers were built to better exploit conventional microprocessors [41, 33, 18].

Specifically with respect to the development of committed-choice logic programming language compilers, a combination of techniques are required to achieve high performance on conventional hardware. Compilation can be viewed as a front-end code generator and a back-end code optimizer. For both sides, abstract machines based on the WAM, e.g., KL1 [17] and Parlog [5], must be discarded in favor of reduced instruction sets that allow fine-grain manipulations. For example, Kliger and Shapiro's decision-graph method for code generation [19] permits fast clause selection as an alternative to conventional indexing methods [2]. Data-flow analysis in the back-end facilitates standard optimizations, such as dereferencing and type-checking elimination, that are impossible in WAM-like instruction sets.

This paper describes the Monaco compiler and runtime system. The compiler translates flat committed-choice logic programs into an abstract machine suitable for execution on shared-memory multiprocessor hosts, targeted to RISC architectures in particular. Our current implementation accepts FGHC programs and produces native Intel 80386 code [16] for the Sequent Symmetry [25]. Monaco's main purpose is to be a testbed for the advanced optimization techniques such as type inferencing [3], mode analysis [40], sequentialization [22], and local memory reuse [32]. Furthermore, the motivation for developing the compiler was to more accurately characterize the parallel execution behavior of concurrent logic programs by avoiding inefficient emulation, a problem in former studies [34].

The paper is organized as follows. Section 2 briefly reviews literature in the area of compilation of concurrent languages. Section 3 outlines the Monaco instruction set and storage model. The compiler is described in Section 4 and the runtime system in Section 5. Performance characteristics are presented in Section 6. Conclusions and future work are given in Section 7.

2 Literature Review

Emulation-based, real parallel, shared-memory implementations of committed-choice languages include Panda and JAM. Panda [28, 35] implements a subset of Flat Guarded Horn Clauses (FGHC) [39], sharing the PDSS compiler [4], and utilizing an abstract machine instruction set similar to WAM [17]. Jim's Abstract Machine (JAM) [5] is a Parlog emulator including support for Or-Parallel execution of *deep* guards. This introduces additional overheads for process management, since a process activation tree must be maintained. We borrowed a great deal from these systems for Monaco's runtime system, since our objective was to examine compilation issues. Neither of these systems uses optimizing compilers.

Strand [10] is also emulation-based and real parallel, although mapped to an intermediate distributed-memory model, allowing portability to alternative hosts. The language is flat Parlog with *assignment*, similar to fully-moded FGHC [40]. The Strand compiler is a commercial product, and thus detailed information is scarce. Strand has a performance advantage when exploiting assignment, but also a potential disadvantage in the overheads incurred when mapping its distributed-memory model onto a shared-memory host. We reconfirm in Section 6 the performance ordering Strand > JAM > Panda. We also demonstrate Monaco's performance relationship to these three systems.

Further restricted from Strand is Janus [27] where the programmer must declare a single producer and consumer for a stream. An experimental Janus-to-C compiler `jc` has been recently developed for uniprocessors [11]. jc has a main advantage over Monaco in that the backend C compiler can do much better register allocation on the host measured in this paper (an 80386-based Symmetry). Like jc, a recent FCP compiler by Kliger is targeted to uniprocessors [18], although it is emulated at present. Monaco's decision-graph compilation method is borrowed from [19] — Monaco extends this locally with dataflow analysis, whereas [18] extends this globally with abstract interpretation to derive procedure bodies optimized for different call sites. Over an extensive set of benchmarks, Kliger reports speedups of 3.2 due to decision graphs over standard indexing, 1.2 due to his global optimizations, and 5.2 due to 68000 native-code compilation [18]. These results encourage us that Monaco is balanced in the sense of putting our effort where the highest payoffs occur.

Finally, we mention RISC-based microprocessor architectures for committed-choice languages: Carmel [13, 12], PIM/i [29], PIM/p [20], and UNIRED-II [31]. These implementations are akin to Monaco, however, they are experiments in specialized hardware, not compilation technology. For example, the macro-call facility in PIM/p is similar in purpose to Monaco's millicode call (Section 4.4), but the hardware implementation is

far more efficient than our interface to 'C'.

3 Monaco Abstract Machine

Committed-choice languages [30] differ from Prolog in several ways, leading to efficient abstract machine definitions that differ from the WAM. First, they have a process-based computation model that does not support backtracking. A computation consists of *reducing* goals (fine-grain tasks or processes), until no unreduced goals remain, in which case the computation succeeds. This implies that fast selection of a committing clause is paramount, as engendered by decision-graph code generation [19]. Second, unification is constrained to be either passive or active. Active unification is more costly than in Prolog because locking is needed to ensure atomic variable binding. Furthermore, to avoid creating circular structures and potential deadlock during multiple unifications of shared variables, we need to create a binding protocol. Third, there is a wide gap in memory-usage efficiency in concurrent and sequential languages. Even parallel Prolog can exploit stacks because of the inherently sequential nature of its threads (e.g., [21, 14]). In committed-choice languages, without sophisticated compiler analysis (e.g., [22]) all goals are potentially concurrent, therefore goal allocation is usually done on a heap. Also, data structures in logic programs are dynamically created and modified, requiring heap storage, although in Prolog, backtracking can naturally reclaim portions of the heap. Overall, the required memory bandwidth of committed-choice languages is significantly greater than that of sequential logic languages. Fourth, process management, i.e., enqueueing, suspending, and resuming operations, must be reduced in frequency and overhead.

The Monaco abstract machine supports seven programmer-visible tagged data types: integers, immediates (symbolic atoms), reference pointers, logical variables, hooked variables, list pairs, and vectors. Abstractly, each of the first five simple objects occupies a machine *cell* consisting of a data field, tag, and lock. As in WAM, a compound object is represented as a pointer to a group of cells containing the list pair or vector. On Symmetry, we implement a cell in eight bytes, with a low-order 3-bit tag in the first word, and the second word currently devoted only to the DYNIX [25] lock. This scheme is similar to that used in Panda, chosen for its simplicity and extensibility.

A logical variable cell is an uninstantiated location, with the data field pointing to the cell itself (as in WAM). When a variable becomes instantiated, its type changes to that of the value that the cell is instantiated with. A *hooked* variable cell is an uninstantiated location on which some process(es) are blocked. The data field of the cell points to a list of suspended processes, as in Panda and JAM. If a hooked cell is

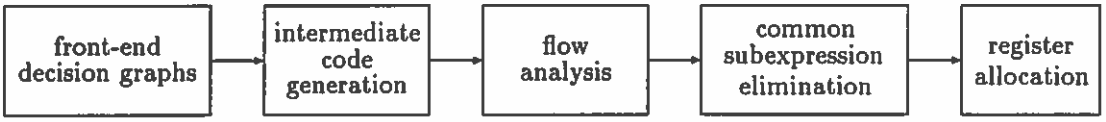

Figure 1: Monaco Compiler Organization (Main Phases Shown)

bound, the processes hooked on the cell must be resumed. Tradeoffs concerning these design decisions are well documented [2, 5].

The Monaco abstract machine is a load-store architecture to facilitate mapping to RISC hosts. Other load-store instruction sets for logic programming languages exist (e.g., [15, 41, 13, 12, 29, 20, 31]) although they have been primarily designed for specialized hardware. Our instruction set is summarized in Table 1. In the table, `Rs`, `Rs1`, and `Rs2` denote source registers, `Rd` denotes a destination register, `N` denotes an integer constant, and `F/A` is the name of an executable procedure.

A majority of the instructions are lightweight and can easily be translated into small sequences of instructions on the host. Most predicate and arithmetic instructions fall into this category. At the other extreme, some instructions are sufficiently complex that not much can be gained by translating them into native code. These are implemented by an interface to the runtime system, called a *millicode call* (see Section 4.4). The middle ground is covered by the data manipulators and constructors. We currently implement these in native code, at some expense in code size, to minimize system-call frequency.

Both active and passive unification are sufficiently complex to warrant implementation in millicode. Full passive unification in the head or guard is only needed in the rare case of shared variables, thus we do not review this algorithm here (see Duvvuru [9]). Active unification commonly occurs in the body, implemented here with the `unify` instruction. In a crop of recent languages, the full power of flat committed-choice languages has been limited, allowing compile-time customization of almost all unification operations. We are particularly interested in moded FGHC (similar to Strand), and have built an associated mode analyzer [22]. With derived mode information, we can compile unifications into cheaper assignments, an optimization we examine in Section 6.2. We discover, however, that without more powerful analysis, the general overloading of logic variables for synchronization causes inefficiencies even for assignment.

4 Compilation

The Monaco compiler is summarized in Figure 1. The pipeline follows a traditional

Instruction	# 80386 instr‡	Semantics
arithmetic and predicates		
`iadd(Rs1,Rs2,Rd)`	3	integer arithmetic (`isub`, `idiv`, `imod`, `imul`)
`iand(Rs1,Rs2,Rd)`	3	bitwise arithmetic (`ineg`, `inot`, `ior`, `ixor`)
`ishl(Rs1,Rs2,Rd)`	3	shift left (`ishra`, `ishrl`)
`ieq(Rs1,Rs2,Rd)`	14?	integer comparison (`ige`, `igt`, `ile`, `ilt`, `ineq`)
`isatom(Rs,Rd)`	9?	type comparison (`isbound`, `ishooked`, `isint`, `islist`, `isnil`, `isref`, `isstruct`, `isunbound`)
`eq(Rs1,Rs2,Rd)`	5	comparison (`neq`)
control		
`br(a,L)`	1	jump to L
`br(Cond,Rs,L)`	2	if (R_s = Cond) then branch to L
`switch(Table,Rs)`	2	jump indirect (on tag(R_s)) through Table
`proceed`	6	complete process
`execute(F/A)`	1	execute process F/A
process management		
`punify(Rs1,Rs2,Rd)`	7†	passive unify
`unify(Rs1,Rs2)`	6†	active unify
`enqueue(Rs)`	5†	push goal on ready queue
`push(Rs)`	8	push address onto suspension stack
`suspend(F/A)`	10†	suspend process F/A
data constructors		
`alloc(N,Rd)`	6	allocate heap by N cells
`ref(Rs,N,Rd)`	3	$R_d := R_s + N$
`initvar(Rs)`	4	mem(R_s) := unbound variable
`initlist(Rs1,Rs2,Rd)`	5	mem(R_d) := pair(R_{s1}, R_{s2}); $R_d :=$ pair ptr
`initstruct(N,Rd)`	11	mem(R_d) := vector of size N; $R_d :=$ vec ptr
`initgoal(N,F/A,Rd)`	12	mem(R_d) := goal record; $R_d :=$ vec ptr
`mkunbound(Rd)`	9	$R_d :=$ ptr to unbound variable
`mkconst(C,Rd)`	1	$R_d := C$ (constant)
`mklist(Rs1,Rs2,Rd)`	11	$R_d :=$ ptr to pair(R_{s1}, R_{s2})
`mkstruct(N,Rd)`	10	$R_d :=$ ptr to vector of size N
`mkgoal(N,F/A,Rd)`	11	$R_d :=$ ptr to goal record
data manipulators		
`move(Rs,Rd)`	2	$R_d := R_s$
`deref(Rs,Rd)`	5	$R_d :=$ dereference of R_s
`car(Rs,Rd)`	3	$R_d :=$ head of R_s
`cdr(Rs,Rd)`	3	$R_d :=$ tail of R_s
`sref(Rs,N,Rd)`	3	$R_d :=$ value of N^{th} slot in vector R_s
`sset(Rs,N,Rd)`	3	N^{th} slot in vector $R_d := R_s$
`ssize(Rs,Rd)`	11	$R_d :=$ size of vector R_s

‡ most frequent path, unless otherwise specified.
? longest path.
† millicode call and return only.

Table 1: Monaco Abstract Instruction Set (Committed-Choice Subset)

organization, where the input is a source program and the output is an equivalent program in the abstract machine instruction set. The front-end parses the program and generates decision graphs with Kliger's algorithm [19]. Alternatively, "don't know" procedures (determinacy testing trees) in languages such as Pandora and Andorra can be generated [37]. These graphs and trees are fed to a code-generation phase which produces rudimentary abstract machine code (assuming an infinite register set).

Flow analysis proceeds by decomposing the program into a flow graph of basic blocks. Type and dereferencing information is propagated through the flow graph in the subsequent phase. At this point, macro-instructions are resolved, redundant computations are recognized and eliminated. Deadcode elimination is a minor pass not show in the figure. Register allocation is performed as the final flow graph optimization. The output from the register allocator is an abstract machine program instantiated with abstract register identifiers.

A series of minor phases (not shown) are then required to perform jump-to-jump short circuiting, dead block removal, branch removal, code flattening, and peephole optimization (in that order). The final output is ready for assembly. Our current assembler translates the Monaco program into 80386 assembly code, and invokes the Symmetry assembler for construction of an object file. The object file can then be linked to the precompiled runtime system, producing an executable. We leverage the Symmetry assembler, requiring this somewhat clumsy linkage protocol, because it is portable (for another host, we target our assembler to the host's assembly language, and ignore intricacies of object image formats).

4.1 Front-End: Decision-Graph Generation

Decision graphs have been shown by Kliger [19] to be effective means of rapidly determining which clause within a procedure can commit. Furthermore, these graphs are space-linear in the number of clauses. We use Kliger's algorithm in our front-end: the specifics are discussed in [37]. For each procedure, a cannonical normalized form is produced. The graph is then generated, where each node is a test (e.g., $X > Y - 3$) and edges are valuations of a test (e.g., yes/no or case values). The leaves of the graph are clause bodies, where we lump *tell* operations with the body. The backend will transform a graph into a sequence of triples suitable for optimization.

The key point here is to compute *residuals* which are clause sets that satisfy a guard (ask) test. Satisfaction requires proving implications between clause constraints and the guard. In general, such proofs are difficult since the domain is unspecified. Furthermore, since multiple residuals may be needed per graph node, this computation is critical to front-end efficiency. Our solution to these concerns is to safely approximate

```
check( X, C, _, NCs, Cs, L, S0, S1 ) :- X = [] |
    append( NCs, Cs, Ps ),
    queen( Ps, [], [C|L], S0, S1 ).
check( X, C, D, _, _, _, S0, S1 ) :-
    X = [P|_], P-C =:= D |
    S0 = S1.
check( X, C, D, _, _, _, S0, S1 ) :-
    X = [P|_], C-P =:= D |
    S0 = S1.
check( X, C, D, NCs, Cs, L, S0, S1 ) :-
    X = [P|Ps], P-C =\= D, C-P =\= D,  D1 := D+1 |
    check( Ps, C, D1, NCs, Cs, L, S0, S1 ).
```

Figure 2: Normalized Procedure `check/8` from *queens*

the proofs by table lookup.[1] For example, the constraint $X > I$ implies the constraint $X > J$ if constant integers $I > J$, for variable X. A set of these relations has been found to be quite effective in allowing optimal graph generation. Complex inferences cannot be made (such as transitivity); however, we have not seen such complexity in typical programs.

Consider the `check/8` procedure in the *queens* program, listed in Figure 2. We chose this example to illustrate both the strengths and weaknesses of decision graph compilation. The first step of processing is to create a normalized canonical form from the source program. This entails flattening the head, pulling all complex terms out into guard "ask" unifications. Furthermore, integer type checks are inserted in the guard for all variables involved in arithmetic expressions, e.g., `integer(P)` is included in clauses 3–5. This allows the indexer to choose type checking early in the decision-graph generation phase, resulting in an efficient graph, shown in Figure 3.

The important points to note in the graph are the indexing choice of switching on the first argument, and the placement of integer type checks for `P`, `C`, and `D`. Switching on the first argument is in fact not optimal when considering *call forwarding*, i.e., shorting callers around operations (such as integer checks in this case) that are known *a priori* from flow analysis. We would like the integer checks as high in the graph as possible. One idea is to force the checks up by inserting additional `integer` guards *in the end-of-recursion case*, although that is neither a satisfying nor automatic solution.

Another flaw in the decision graph is that the path to clause two (node 2) requires *three* arithmetic inequalities. Ideally, node (1) should commit to clause two. The problem lies in the power of our inferencing mechanism computing residuals. A clause is not placed in the residual of a branch test unless a guard in that clause *implies* the

[1] The Aquarius Prolog compiler [41] simplifies formulae in a similar manner. Both Monaco and Aquarius use about 50 rules of comparable complexity.

```
switch(X,
  [case([],commit(clause 1))
   case('.'/2,
      ask(integer(P),
          yes(ask(integer(C),
                  yes(ask(D),
                          yes(ask((C-P =\= D),
                                  yes(ask((P-C =\= D),
                                          yes(commit(clause 4))
                                          no(go(7))                    (1)
                                        7:other(go(6)))),
                                  no(commit(clause 3)),
                                6:other(
                                    ask((P-C =:= D),
                                        yes(commit(clause 2)),         (2)
                                        no(go(8)),
                                      8:other(go(5))))))),
                          no(go(5)),
                        5:other(go(4)))),
                no(go(4)),
              4:other(go(3)))),
          no(go(3)),
        3:other(go(2)))),
  2:default(go(1))]
1:suspend
```

Figure 3: Stylized Decision Graph for `check/8`

test (thus retaining space linearity). Actually, $P - C = D$ *does* imply $C - P \neq D$, but we have refrained from adding such inferencing smarts to the compiler. Our inferences are purely table driven, and although we could insert many more table entries for such ad hoc cases, a general prover would be best. A final point: the rather circuitous routes to suspension are entirely collapsed by dataflow analysis and jump-chain shorting in subsequent phases of the compilation.

4.2 Code Generation

Code generation translates the decision graphs into intermediate Monaco instructions. The translation algorithms are straightforward, perhaps naive concerning arithmetic expressions. Key to code generation is a "one register — one value" invariant that facilitates all later phases, e.g., a register can be defined only once within a procedure. Perhaps the trickiest part of code generation is the construction of switch-on-tag statements, which are composed of eight binary switch instructions, each in its own basic block. This allows us to standardize basic blocks in a binary tree. Body generation follows the standard style (e.g., [17]) of enqueueing all body goals but the first, which is executed immediately.

Subsequent dataflow analysis proceeds from the Monaco code, driven by a table describing each instruction's operand uses and definitions, as described in the next section. In retrospect, we found that certain analyses, such as call forwarding [6], is best done on the decision graph, not the generated code. Although dataflow information must be derived earlier to do this, it is much easier to rearrange portions of the flow graph, with no concern for register bindings.

4.3 Flow-Analysis Optimizations

Data-flow analysis is fundamental to most of the compiler optimizations, to the point where the preliminary code is particularly naive and *requires* flow analysis to clean it up. We took this approach to keep the compiler modular, although it impacts compile time. The basis for flow analysis is the construction of a flow graph of basic blocks from the preliminary code. A standard construction algorithm is used [1] with the note that Monaco instructions such as **execute**, **proceed**, and **suspend** represent control transfers, and thus terminate blocks.

Note that the program is analyzed locally, i.e., on a procedure-by-procedure basis, to perform common subexpression elimination (CSE). A flow graph, created for a procedure, is topologically sorted to ensure that all ancestors of a child block are analyzed before the child is analyzed. Type information is then collected and propagated from the root, taking the set intersection (as the least upper bound) of information arriving to a child node from its ancestors. The information is essentially an association list matching pseudo-registers and their abstract contents. For example, suppose we know `R3 = car(deref(R1))`, and we encounter a Monaco instruction `deref R3,R4`, then we derive the new information: `R4 <- deref(car(deref(R1)))`. A common association entry is `R6 <- R5`, created from the instruction `move R5,R6`.

Code within the blocks is rewritten on the fly during analysis to share common subexpressions. The most common case of this is shorting moves, e.g., `move R5,R6` followed by `isint R6,R7`, will rewrite the latter instruction to be `isint R5,R7`. Later deadcode elimination is of course required here. Branch conditions are also propagated throughout the flow graph. We must be careful to distinguish the taken condition from the not-taken condition. For example, entering some block in the graph, we may know that `integer(deref(car(deref(R1))))` must be true, by consideration of the flow to that point. This allows branch shorting, e.g., `br nz,R8`, which branches if `R8` is not zero, can be combined with flow information `R8 <- isint(deref(car(deref(R1))))` and the previous branch information, to derive that the branch is always taken, and so it will be rewritten as an unconditional jump.

Deadcode elimination and register allocation require further flow graph analysis.

Live-range analysis [1] is performed on the graph, producing the information of which registers are live within each basic block. We chose to collect this information at block granularity rather than instruction granularity to reduce compile time. Large clause bodies can have relatively large basic blocks, reducing the effectiveness of this technique. For this reason we artificially split such blocks at each body goal. In general, this heuristic is sufficient to retain accuracy, as discussed in Section 6.

If a value is never used, then its live range will be empty. Hence, deadcode elimination is performed by removing instructions containing values with empty live ranges. Give the previous analysis, this phase is trivial. Next, we allocate registers locally to each procedure.[2] The local allocation method used is based on the liveness of the registers, and is performed on a basis-block granularity to match live-range analysis. The most live name (i.e., having the most basic blocks over its life) is allocated first, and so on. The algorithm is non-backtracking, so lack of an available register for the next most frequent name requires generation of spill code. The spill is allocated to a vector local to the procedure.

The goodness of such a naive scheme relies on the accuracy of flow analysis. Allocating on a block basis can lead to frequent spill code, although this is alleviated by splitting the body along individual goals. Still, a goal requiring the evaluation and loading of many actuals is the most likely to cause spills. Spilling can be reduced by artificially splitting the blocks into finer grains (in the limit: instructions). Because we were targeting our initial experiments to an 80386 backend with so few registers available to the program, it became hopeless to avoid frequent spilling. Instead we generate memory accesses to a pseudo-register array. This array is small enough to easily fit in cache, yet large enough to avoid spilling. This stop-gap measure allows us to refine the compiler for eventual porting to a host with sufficient registers.

Our allocator averages 7.3 registers for the 37 benchmark procedures considered in Section 6. Trivial procedures ranged from 2–7 registers, whereas more complex procedures ranged from 11–17 registers. Future work includes implementing a global allocator based on cooperation across procedure call boundaries.

Final phases include shorting of all jump chains, which leads to dead basic blocks that are removed. The flow graph is then flattened and a peephole optimizer filters the code stream. Currently the peepholer is limited: its primary function is to attempt common subexpression elimination (CSE) of spill sequences. This is inherent to the well-known problem of where to allocate registers: before or after CSE. Since we allocate before, we cannot eliminate spill code redundancies. The peepholer cannot make much

[2] Global register allocation in concurrent languages such as FGHC first requires sequentialization of threads, which we are pursuing[22].

headway here, and we plan to explore a split register allocator, as in gcc, to better solve the problem.

We translate the Monaco code into 80386 Symmetry assembly code in a final assembler module. The 80386 code is reassembled with the Symmetry assembler, producing an linkable object file. The object is linked directly to the runtime system object, forming a executable file (customized to this particular source program). This method is quick and simple — the system object need only be linked, not recompiled. Only objects are saved — executables are deleted after use. This is sufficient for our use as an experimental testbed.

4.4 Example

The Monaco code for the quicksort procedure written in FGHC, is shown in Figure 4 (`part/4` not shown). The precise semantics of the instruction set are not necessary to follow the example. Before entry to a procedure, the input arguments are loaded in to the abstract machine registers numbered from zero. The entry point to this procedure is labeled `qsort/3`. Arguments are dereferenced before they are tested (1). Quicksort dispatches on its first (input) argument in a switch on tag instruction (2). If unbound, control flows to L1, where the argument is pushed on the suspension stack and suspension occurs (3–4). If an immediate or vector, control flows to L2 where the suspension routine is called with no pushed arguments (4), resulting in failure. If nil, control flows to L3, where the argument is compared to nil and either fails or commits to the first clause (5–9). If the first argument is a list, the second clause commits at L4 (10–35). The `alloc` instruction (10) allocates 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 `part/4` procedure, after setting up its arguments in appropriate registers, exploiting last-call optimization. We will comment more about how this code is generated in the next section.

There are several interesting points about how code is generated from the decision graph. These characteristics are analyzed empirically in Section 6.

- Originally binary branches were favored over switches, because we planned Monaco for a RISC backend. However, the 80386, based on a condition code register is not conducive to branching on conditions stored in registers. Therefore we introduced the `switch` instruction, assembled into an indirect jump-on-tag table.

- Structures are created on the heap by allocating space for the entire clause body, and then filling it in for each structure (including goal records). For each structure

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

qsort/3:  deref    r0,r4          % r4 := deref(Argument 1)          1
          switch   [L1,L4,L3,L2,L2,L1,L1,L1],r4                      2
L1:       push     r0             % No. Push Argument 1. Suspend.   3
L2:       suspend  qsort/3        % Suspend                         4
L3:       mkconst  [],r3          % Argument 1 is []?               5
          eq       r4,r3,r5                                         6
          br       z,r5,L2        % No, fail.                       7
          unify    r1,r2          % Yes. Rest = Ans                 8
          proceed                                                   9
L4:       alloc    15,r5          % Allocate 15 cells on heap.      10
          move     r5,r6                                            11
          initgoal qsort/3,r6     % r6 -> goal qsort/3              12
          ref      r5,5,r3        % r3 := L                         13
          initvar  r3                                               14
          sset     r3,1,r6        % 1(r6) := L                      15
          ref      r5,6,r8        % r8 := Y1                        16
          initvar  r8                                               17
          sset     r8,2,r6        % 2(r6) := Y1                     18
          sset     r2,3,r6        % 3(r6) := T                      19
          enqueue  r6             % enqueue qsort(L,Y1,T)           20
          ref      r5,7,r9                                          21
          initgoal qsort/3,r9     % r9 -> goal qsort/3              22
          ref      r5,12,r2       % r2 := S                         23
          initvar  r2                                               24
          sset     r2,1,r9        % 1(r9) := S                      25
          sset     r1,2,r9        % 2(r9) := Y                      26
          car      r4,r1          % r1 := X                         27
          ref      r5,13,r12                                        28
          initlist r1,r8,r12      % r12 := [X|Y1]                   29
          sset     r12,3,r9       % 3(r9) := [X|Y1]                 30
          enqueue  r9             % enqueue qsort(S,Y,[X|Y1])       31
          cdr      r4,r0          % r0 := R                         32
          execute  part/4         % call part(R,X,S,L)              33
```

Figure 4: Compiled Quicksort Procedure in Monaco

argument, a reference is built (with `ref`) pointing to the heap location where it will be constructed, and the storage is initialized (e.g., `initvar` and `initgoal`). Then the structure argument position is bound to the reference (with `sset`). This implies that variables reside outside of structures, which will ease future optimizations concerning local memory reuse [32].

- The final Monaco code shown in Figure 4 is the result of extensive dataflow analysis and optimization. With no optimizations the procedure has 43 Monaco instructions, an increase of 30%. The primary savings in code size, and correspondingly execution time, comes from CSE and tight control.

- The compiler was purposely kept as elegant and modular as possible, with a minimum number of stages, and therefore less than optimal code is produced. For example, we chose to do register allocation *after* CSE. Thus if we have register spilling, the spill sequences will not be simplified. We also chose to abide by a code generation policy wherein every pseudo register is defined at most once per procedure. Given this policy, which facilitates dataflow analysis, we need to assign *different* pseudo registers to passed parameters and tail-recursive arguments. Those registers need to be coerced during register allocation. The result is poor allocation in the basic block containing the tail recursion. Instead of more dataflow analysis to solve this [26], a patch was added to scan the block in reverse, shorting out register-to-register moves. This produces reasonable, but not excellent, code.

The macro assembler is the penultimate stage in the compilation process. It translates a Monaco assembly program into an 80386 assembly program, for final translation by the DYNIX assembler *as*. Figure 5 gives two examples of macro expansion of Monaco into 80386 assembly. Instructions `unify`, `punify`, `enqueue`, `suspend`, and `execute` are sufficiently complex to warrant implementation within the runtime system as millicode. As shown for `punify`, a millicode call is accomplished by means of an indirect jump table, with parameters passed as in 'C'.

5 Monaco Runtime System

In this section we give a brief overview of the runtime kernel to put the subsequent performance analysis in perspective. For a detailed discussion, see Tick [36].

Concurrent logic languages, due to the nature of the fine-grain tasks, single-assignment variables, and heap-based memory management, require extensive runtime support. The main objectives of the runtime system are to support fast process man-

```
        movl    Rs*4(REGS),TMP0     # deref(Rs, Rd)
        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

                                    # punify(Rs1, Rs2, Rd)
        pushl   Rs2*4(REGS)         # push Rs2 on stack
        pushl   Rs1*4(REGS)         # push Rs1 on stack
        pushl   $LM_RET             # push return address
        jmp     *PUNIFY_OFF(REGS)   # indirect call to punify
LM_RET: addl    $8,%esp             # readjust stack
        movl    TMP0,Rd*4(REGS)     # Rd := punify(Rs1,Rs2)
```

Figure 5: Examples of 80386 Macro Expansion on Symmetry (TMP0, TMP1 and REGS are machine register aliases)

agement, memory allocation, and runtime system calls, as described in this section. Like most WAM-based systems (e.g., [21, 28, 5]), the Monaco kernel is based on "task farming:" a set of *worker* processes that run as regular DYNIX processes on all the processors. A worker process corresponds to one Monaco abstract machine processor and operates in a loop: looking for reducible goals in the pool, finding one and reducing it, adding the body goals of that clause to the pool, and continuing to look for more work.

To reduce synchronization, the goal pool is split among the workers — each has a private goal queue, implemented as a list. Thus we both enqueue and dequeue goals from the queue head, resulting in pseudo depth-first execution (not pure because idle workers steal goals from busy worker's queues). Depth-first evaluation has been shown to be relatively efficient [28], so we adopted it here. We also adopted the JAM scheduling strategy of a ring: an idle worker examines the private queues of successive clockwise neighbors until a goal is found. This scheme is efficient because the idle worker responsible for finding work, affecting busy workers only when synchronizing on the queue.[3] This scheme works effectively and produces reasonable load balancing, as shown in JAM and here (see Section 6). Our current research emphasis is in an efficient instruction set and compilation procedure, so we have adopted standard techniques for process management. Suspension and resumption is done with logical variables

[3] JAM reduces synchronization by stealing from the tail of the queue. We did not implement this optimization because we plan to use priority queues exploiting granularity estimations [38] in future versions of Monaco.

hooked to suspended goals via suspension slips, as in JAM Parlog [5] and KL1 [17] implementations.

Most goals allocate heap memory for creating data and body goal structures. We reduce the frequency of allocation by aggregating all heap requests within a clause into a single millicode call. The Monaco heap is split into equal-sized slices for each worker, avoiding synchronization. Each heap is initialized with two range pointers delimiting the unallocated area. All heap allocations, approximately one per reduction, manipulate and check these pointers accordingly. The overflow check is sufficiently frequent that we chose to generate it in native code. This naive heap implementation is sufficient for our initial evaluation of compilation techniques. The current version of Monaco does not yet implement garbage collection (and so collection overheads are not accounted for in the empirical analysis given in Section 6).

6 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 Strand (Buckingham) [10], JAM V1.4 [5], PDSS V2.52.19 [4], Panda (using PDSS V0.8 compiler!) [28], Janus [11], and SICStus Prolog V2.1. For multiprocessor execution, comparison is made only with JAM because it is the fastest of the parallel systems with comparative scheduling. Empirical evaluation was conducted on a Sequent Symmetry S81 system with 16 MHz Intel 80386 microprocessors. Execution times (in milliseconds) are calculated as the elapsed time between starting the computation, until all the processors become idle. All measurements presented are the minimum of several runs.

Table 2 shows uniprocessor (one PE) execution times, normalized to Monaco, for a benchmark suite (listed in the Appendix). Monaco is compiled with full optimizations, but without mode analysis (this primarily affects *queens* as mentioned below). The measurements are only an approximate comparison: the systems offer different facilities (e.g., language capabilities, types of garbage collectors) that make a fair comparison difficult. For example, PDSS, Janus, and SICStus are sequential systems without any multiprocessor overheads. Furthermore, the algorithms used, e.g., *queens*, may not be optimal for Prolog. In any case, these measurements satisfy us that the Monaco has slightly better performance than alternative multiprocessor systems.

We still have a far way to improve (10%–500%) to achieve Janus speeds (of course, we can make this up with a multiprocessor). With respect to Strand, the fastest com-

Program	Monaco	Strand	JAM	Panda	PDSS	Janus	SICStus
hanoi(14)	1.0	1.3	1.7	2.1	2.1	0.27	1.1
nrev(1000)	1.0	1.8	2.0	3.3	2.9	0.20	1.1
qsort(1000)	1.0	1.9	2.4	5.2	2.7	0.60	1.8
queens(10)	1.0	0.59	3.2	4.6	4.2	0.91	2.4
primes(5000)	1.0	3.0	3.7	5.8	3.7	0.48	2.3
pascal(200)	1.0	2.4	2.3	4.4	2.6	0.42	1.6
geo. mean	1.0	1.6	2.5	4.0	3.0	0.43	1.6

Table 2: Uniprocessor Execution Time Normalized to Monaco on Sequent Symmetry (Full Optimizations)

petition publicly available, we are running from 40% slower to three times faster. We believe the 40% gap in *queens* is due to 1) a non-optimal decision graph, as described in Section 4.1; 2) our somewhat naive code generation of arithmetic expressions (no static type inferencing is done); and 3) no exploitation of mode information (to strength reduce variable-to-variable unifies, which comes naturally in Stand). We discuss optimizations in Section 6.2 to reduce (2) and (3), resulting in 20% speedup in *queens* which reduces the gap with Strand to −30%.

The multiprocessor execution times of Monaco and JAM are given in Table 3. The speedups, shown in Table 4, are calculated as the ratio of execution times of the same multiprocessor system on one PE to multiple PEs. Thus the metric is biased, but still gives us a first approximation of the utility of exploiting parallelism. Monaco and JAM use 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 as in JAM. Thus we are not surprised that speedups are similar. It is not clear why JAM's execution time *increased* from 4 to 8 PEs on *qsort*, the outlier of the benchmarks. In general we expected the experiment to be biased to JAM which allocates proportionally more heap with increasing numbers of PEs. Again, these are preliminary measurements and we conclude simply that Monaco is successfully exploiting fine-grain parallelism, even though its streamlined abstract machine achieved gains in absolute performance.

6.1 Execution Profiling

We instrumented the Monaco runtime kernel to measure the performance of the frequently-used millicode procedures: `unify`, `enqueue`, and `suspend`. All the measurements were taken with one worker, using tools built from UNIX *profil* and *prof*. Table 5 shows the percentage execution time spent in the user program (Monaco code) and the runtime kernel. The kernel consists of **Monitor** (where the worker steals or de-

Program	1 PE	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco (msec)					
hanoi	4,364	2,339	1,218	722	598	558
nrev	19,153	11,740	6,136	3,394	2,503	2,155
qsort	21,412	12,970	6,621	3,496	2,438	1,954
queens	43,305	30,348	12,413	6,153	4,041	3,082
primes	12,841	7,549	3,788	1,984	1,390	1,117
pascal	8,973	4,981	2,499	1,294	919	724
	JAM (msec)					
hanoi	7,360	3,780	2,260	1,620	1,390	1,290
nrev	38,030	33,550	10,200	5,290	3,570	2,700
qsort	50,900	26,300	14,120	19,150	15,220	13,170
queens	139,590	69,860	34,900	17,490	11,680	8,780
primes	48,120	24,930	12,520	7,300	6,270	5,550
pascal	20,300	10,130	5,140	2,620	1,780	1,360
	JAM/Monaco					
hanoi	1.7	1.6	1.9	2.2	2.3	2.3
nrev	2.0	2.9	1.7	1.6	1.4	1.3
qsort	2.4	2.0	2.1	5.5	6.2	6.7
queens	3.2	2.3	2.8	2.8	2.9	2.8
primes	3.7	3.3	3.3	3.7	4.5	5.0
pascal	2.3	2.0	2.1	2.0	1.9	1.9
geo. mean	2.5	2.3	2.3	2.7	2.8	2.8

Table 3: Multiprocessor Performance of Monaco (Optimized) and JAM

Program	2 PE	4 PE	8 PE	12 PE	16 PE	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco					**JAM**				
hanoi	1.9	3.6	6.0	7.3	7.8	1.9	3.3	4.5	5.3	5.7
nrev	1.6	3.1	5.6	7.7	8.9	1.1	3.7	7.2	10.7	14.1
qsort	1.7	3.2	6.1	8.8	11.0	1.9	3.6	2.7	3.3	3.9
queens	1.4	3.5	7.0	10.7	14.1	2.0	4.0	8.0	12.0	15.9
primes	1.7	3.4	6.5	9.2	11.5	1.9	3.8	6.6	7.7	8.7
pascal	1.8	3.6	6.9	9.8	12.4	2.0	3.9	7.7	11.4	14.9
geo. mean	1.7	3.4	6.3	8.8	10.7	1.8	3.7	5.7	7.6	9.3

Table 4: Multiprocessor Speedups on Monaco and JAM

			Kernel						
					Goals		Unify		
Program	**User**	**Kernel**	**Monitor**	**Run**	**Enq**	**Susp**	**Stub**	**Rest**	**Misc**
hanoi	47.11	52.89	9.10	16.08	9.47	0.00	11.06	0.00	7.15
nrev	56.24	43.76	0.39	0.33	0.11	0.00	34.61	0.20	8.12
qsort	62.36	37.64	0.39	0.28	0.28	0.00	30.59	0.01	6.11
queens	54.48	45.52	8.84	12.46	8.82	0.00	7.90	0.00	7.47
prime	69.74	30.26	0.53	0.18	0.10	0.00	24.01	0.01	5.34
pascal	54.63	45.37	2.16	2.72	2.61	0.00	29.25	0.10	8.61
arith. mean	57.43	42.57	3.57	5.34	3.57	0.00	22.90	0.05	7.13
geo. mean	57.00	41.94	1.55	1.44	0.94	0.00	20.05	0.00	7.04

Table 5: Percentage Execution Times in Kernel and User Program (Fully Optimized, on One Worker)

queues goals and prepares them for execution), **Run** (a stub currently needed to call the Monaco code), **Enq** (millicode for enqueueing goals), **Susp** (millicode for suspending goals), **Stub** (the fast case of unification), **Rest** (the general body of unification), and miscellaneous. Suspension is negligible in these benchmarks, making **Susp** and **Rest** insignificant. *Hanoi* and *queens* enqueue a large number of goal records and hence the time spent in **Enq**, **Monitor**, and **Run** is significant: on average 32.39% compared to 12.48% over all the programs.

The statistics indicate that a more streamlined thread interface will be beneficial for procedure intensive programs. For example, the **Run** stub is a patch around some compilation difficulties, and in theory is not needed. With a RISC backend, we would avoid loading goal arguments into memory-simulated registers, as we currently do for the 80386, thus reducing **Monitor** time. More far-reaching solutions involve increasing granularity by sequentialization (see next section). A further kernel optimization would be to pull up the unification **Stub** (now averaging 22.9%, the most overhead of any single function) into assembly code.

6.2 Compiler Optimizations

Table 6 shows the performance gains of three optimizations: **CSE**: dataflow analysis enabling common subexpression elimination and subsequent deadcode removal, **switch**: using branch tables for generating switch nodes with the decision graphs, and $X := Y$: the previously described variable-to-variable unification optimization. Each effect is shown as a speedup relative to the previous compilation. In the case of switches, the default method uses a chain of binary branches. In the case of unification, the default

Program	CSE	switch	$X := Y$
hanoi(14)	2.19	5.01	0.65
nrev(1000)	16.06	8.48	−0.50
qsort(1000)	32.21	16.88	−0.22
queens(10)	18.07	12.75	11.66
prime(5000)	27.51	16.61	−0.14
pascal(200)	21.49	12.40	1.44
geo. mean	15.15	11.11	—

Table 6: Percentage Performance Gain with Dataflow Analysis, Branch Tables, and Mode Analysis Optimizations

method uses a stub that cannot quickly perform variable-to-variable assignment (for fear of deadlock in nonmoded programs). The measurements clearly demonstrate the effectiveness of dataflow analysis, and to a lesser extent, fast branching. Note that discounting the outlier *hanoi*, CSE accounts for 22% average gain. The assignment optimization benefits only *queens*, the one benchmark that collects multiple solutions in a difference list, and thus shorts the list frequently with variable-to-variable unification. The performance gain/loss for the other benchmarks is insignificant, so we do not calculate a mean.

To experiment with the utility of decision graph code generation, we did not have an alternative frontend that used WAM-style indexing. Instead, we generated decision graphs with a naive indexer. The naive indexer simply chose the next available test for the next node. However, since arithmetic comparisons required that integer types be checked earlier, even the naive indexer chose integer tests first. The benchmark suite was sufficiently simple that the naive indexer produced code that was approximately as fast as the sophisticated indexer. More complex procedures are required to demonstrate a significant difference, e.g., [18].

Three additional optimizations were consider by hand modification of the compiler-generated code for *queens*: optimal register allocation, efficient branching, and efficient calling. The results are given in Table 7. To experiment with the performance loss due to register allocation, we hand-allocated the Monaco registers for *queen*. Compiler register allocation averaged eight registers per procedure, ranging from 4–15. Hand allocation reduced this to 5.6 registers per procedure, ranging from 2–12 (this included optimistically rearranging instructions to ease allocation). Because these registers are simulated, global allocation has no beneficial effect. The only advantage we can measure is the reduction of move instructions, which as seen has little (0.8%) speedup. Hand allocation of 80386 registers was too intricate to contemplate, even for *queens* with 803

Experiment	Time (msec)	% Gain
baseline (standard opts)	38,578	
hand register allocation	38,275	0.79
efficient 80386 branching	37,165	2.99
call forwarding	36,136	2.85
total		6.62

Table 7: Additional Optimizations, Measured for Queen(10)

assembly instructions.

The poor match between Monaco's branches and the 80386 was alleviated by a peephole optimizer in the assembler. The problem with this method is that it squashes the branch-condition register, erroneous if that register is live. This optimization is safe for *queens*, allowing us to measure its impact, a 3% speedup.

Call forwarding is an optimization wherein each tail-recursive caller jumps directly to the portion of the callee that is relevant, skipping work that is known to be superfluous. We hand coded this for *queens*, the benchmark program with the most to gain from the optimization, by modifying the intermediate Monaco program. The 2.8% speedup we measured can be calibrated with significantly higher reported gains in [6] by considering that 42% of our execution time is spent in the runtime kernel, not present in Janus. Thus our results might be normalized upwards to 5%.

As a final experiment, we developed an alternative backend for the Monaco compiler which translates FGHC into sequential C code. The main advantages are exploiting the C compiler's 80386 register allocator, and avoiding fine-grain task management overheads. The enabling technology for this is mode and sequentialization analysis [22]. Essentially, fully-moded FGHC programs can be analyzed to determine all path modes nonambiguously, allowing us to impose an ordering on clause body goals that can be sequentially executed, safe from internal deadlock (external producers may still cause suspension). Efficient code can be generated for such programs, illustrated in Table 8 for the *qsort* benchmark. The row labeled "sequential C" is generated by our alternative backend. For instance, the code is $10.5/2.2 = 4.8$ times faster than the standard Monaco compiler. The details of the code generation are given in Massey and Tick [22], and are slightly biased because 1) 32-bit cells rather than 64-bit cells (as in Monaco) were used; 2) suspensions due to external sources cannot be handled. In any case, we are encouraged to further develop this backend so that our standard multi-threading can coexist with sequentialized threads.

System	PEs	Problem Size	Time (sec)
PDSS	1	125	1.4
		250	5.2
		500	20.5
Monaco	1	500	10.5
	4	125	0.20
		250	0.75
		500	2.9
	8	500	1.5
sequential C	1	125	0.14
		250	0.58
		500	2.2

System	PEs	Problem Size	Time (sec)
Strand	1	500	10.8
JAM Parlog	1	500	13.1
handcrafted C	1	125	0.10
		250	0.40
		500	1.5

Table 8: Performance of Sequentialized and Parallel QuickSort

7 Conclusions

We have presented the Monaco runtime kernel and compiler, a shared-memory implementation of flat committed-choice languages. In the spirit of Van Roy [41] and Taylor [33], the key design decision was to move from a WAM-based to a lower level intermediate instruction set. This demanded the construction of an optimizing compiler based on local dataflow analysis. Our system is unique in that, 1) it translates concurrent programs onto a parallel execution model; 2) it produces intermediate code targeted for high-performance on RISC hosts; 3) backend generates native code; 4) mode analysis allows the conversion of unifications into assignments in fully-moded programs, and 5) it forms a foundation for global optimizations that can then be accurately measured within a streamlined system. We have presented empirical measurements characterizing the execution profile of the system, demonstrating the utility of the optimizations, and indicating areas for future gains. In summary, Monaco ran at least 1.6 times faster, yet achieved slightly higher speedups, than comparable parallel systems. Still, it runs 2.4 times *slower* than `jc` compilation into sequential 'C'.

Future work involves tuning the kernel and 80386 backend, building a RISC backend, and also a 'C' backend. The latter two promise significantly better register allocation, and producing 'C' will make the compiler portable. We are starting to integrate global optimization techniques within Monaco: sequentialization (underway), local reuse analysis [32], weighted decision graphs [7], and call forwarding [6].

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. L. Hansen designed the original Monaco instruction set. M. Korsloot wrote the indexer of the Monaco compiler. S. Duvvuru completed the first implementation of the Monaco runtime system. F. Rakoczi implemented the Monaco profiling tools. The authors thank B. Massey for his kind assistance in this research.

References

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

[2] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[3] M. Bruynooghe, A. Marien, and G. Janssens. The Impact of Abstract Interpretation: and Experiment in Code Generation. In *International Conference on Logic Programming*, pages 33–47. Lisbon, MIT Press, June 1989.

[4] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System PIMOS. In *International Conference on Fifth Generation Computer Systems*, pages 230–251, Tokyo, November 1988. ICOT.

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

[6] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan. Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. Dept. of Computer Science, University of Arizona, December 1992.

[7] S. Debray, S. Kannan, and M. Paithane. Weighted Decision Trees. In *Joint International Conference and Symposium on Logic Programming*, pages 654–668. Washington D.C., MIT Press, November 1992.

[8] T. P. Dobry, A. M. Despain, and Y. N. Patt. Performance Studies of a Prolog Machine Architecture. In *International Symposium on Computer Architecture*, pages 180–190. Boston, IEEE Computer Society Press, December 1985.

[9] 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.

[10] 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.

[11] 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.

[12] A. Harsat and R. Ginosar. CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog. *New Generation Computing*, 7:197–218, 1990.

[13] A. Harsat and R. Ginosar. CARMEL-4 The Unify-Spawn Machine for FCP. In *International Conference on Logic Programming*, pages 840–854. Paris, MIT Press, June 1991.

[14] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[15] B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, A. M. Despain W. R. Bush, J. M. Pendleton, and T. Dobry. Fast Prolog with an Extended General Purpose Architecture. In *International Symposium on Computer Architecture*, pages 282–291, Seattle, June 1990. IEEE Computer Society Press.

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

[17] 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.

[18] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992.

[19] 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.

[20] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa, and K. Hirano. Architecture and Implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, pages 414–424, Tokyo, June 1992. ICOT.

[21] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830, Tokyo, November 1988. ICOT.

[22] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, St. Petersburg, July 1993. Submitted.

[23] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *International Symposium on Logic Programming*, pages 104–113. San Francisco, IEEE Computer Society Press, August 1987.

[24] R. Nakazaki *et al.* Design of a High-Speed Prolog Machine (HPM). In *International Symposium on Computer Architecture*, pages 191–197. Boston, IEEE Computer Society Press, June 1985.

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

[26] F. Rakoczi, K. Peery, and G. Folkestad. Optimizing the Monaco Compiler. Dept. of Computer Science, University of Oregon, June 1992. Unpublished.

[27] 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.

[28] 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.

[29] M. Sato, K. Kato, K. Takeda, and T. Oohara. Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine. Technical Report TR-676, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1991.

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

[31] K. Shimada, H. Koike, and H. Tanaka. UNIRED-II: The High Performance Inference Processor for the Parallel Inference Machine PIE64. In *International Conference on Fifth Generation Computer Systems*, pages 715–722, Tokyo, June 1992. ICOT.

[32] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508. Washington D.C., MIT Press, November 1992.

[33] 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.

[34] E. Tick. A Performance Comparison of AND- and OR-Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 452–470. Lisbon, MIT Press, June 1989.

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

[36] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe.* Springer Verlag, June 1993.

[37] E. Tick and M. Korsloot. Determinacy Testing for Nondeterminate Logic Programming Languages. *ACM TOPLAS*, 1993. In press. Also available as University of Oregon Technical Report CIS-TR-93-07.

[38] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11(3–4), June 1993. Also available as University of Oregon Technical Report CIS-TR-91-19a.

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

[40] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 1993. In press.

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

Appendix: Benchmark Source Code Listing (FGHC)

```
:- module suite.
:- public nrevgo/1, qsortgo/1, hanoigo/1.
:- public queengo/1, primego/1, pascalgo/1.


nrevgo(   N ) :- N > 0 | gen( N, X ), nrev( X, _ ).
qsortgo(  N ) :- N > 0 | gen( N, X ), qsort( X, _ ).
hanoigo(  N ) :- N > 0 | hanoi( N, _ ).
queengo(  N ) :- N > 0 | queen( N, _ ).
primego(  N ) :- N > 0 | prime( N, _ ).
pascalgo( N ) :- N > 0 | pascal( N, _ ).


gen( N, L ) :- true | gen( 0, N, L ).

gen( K, N, L ) :- K >= N | L = [].
gen( K, N, L ) :- K <  N | K1 := K+1, L = [K|Ls], gen( K1, N, Ls ).

nrev( [], Y ) :- true | Y = [].
nrev( [A|X], Y ) :- true | nrev( X, T ), append( T, [A], Y ).

append( [], X, Y ) :- true | Y = X.
append( [A|B], X, Y ) :- true | Y = [A|Z], append( B, X, Z ).


qsort( X, Y ) :- true | qsort( X, Y, [] ).

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 ).

partition( [X|Xs], A, S, L ) :- A < X | L = [X|L1],
    partition(Xs ,A,S,L1).
partition( [X|Xs], A, S, L ) :- A >= X | S = [X|S1],
    partition( Xs, A, S1, L ).
partition( [], A, S, L ) :- integer(A) | S = [], L = [].


hanoi( N, X ) :- true |
    move( N, left, center, right, Y, [] ),
    count( Y, X ).

move( 0, _, _, _, O1, O2 ) :- true | O1 = O2.
move( N, A, B, C, O1, O4 ) :- N>0, M := N-1 |
    move( M, A, C, B, O1, O2 ),
    O2 = [ A-B | O3 ],
    move( M, C, B, A, O3, O4 ).

count( L,N ) :- true | count( L,0,N ).

count( [], M, N ) :- integer(M) | N = M.
count( [_|Xs], M, N ) :- M1 := M+1 | count( Xs, M1, N ).
```

```
queen( N, M ) :- true |
    gen( N, L ),
    queen( L, [], [], A, [] ),
    count( A, M ).

queen( [C|Cs], NCs, L, S0, S2 ) :- true |
    check( L, C, 1, NCs, Cs, L, S0, S1 ),
    queen( Cs, [C|NCs], L, S1, S2 ).
queen( [], [], L, S0, S1 ) :- true | S0 = [L|S1].
queen( [], [_|_], _, S0, S1 ) :- true | S0 = S1.

check( [], C, D, NCs, Cs, L, S0, S1 ) :- integer( C ), integer( D ) |
    append( NCs, Cs, Ps ),
    queen( Ps, [], [C|L], S0, S1 ).
check( [P|_], C, D, _, _, _, S0, S1 ) :- P-C =:= D | S0 = S1.
check( [P|_], C, D, _, _, _, S0, S1 ) :- C-P =:= D | S0 = S1.
check( [P|Ps], C, D, NCs, Cs, L, S0, S1 ) :-
    P-C =\= D, C-P =\= D, D1 := D+1 |
    check( Ps, C, D1, NCs, Cs, L, S0, S1 ).



prime( M,  N ) :- true |
    prime1( M, Ps ),
    count( Ps, N ).

prime1( Max, Ps ) :- true |
    gen( 2, Max, Ns ),
    sift( Ns, Ps ).

sift( [P|Xs1], Zs0 ) :- true | Zs0 = [P|Zs1],
    filter( Xs1, P, Ys ),
    sift( Ys, Zs1 ).
sift( [], Zs0 ) :- true | Zs0 = [].

filter( [], _, Ys ) :- true | Ys = [].
filter( [X|Xs], P, Ys ) :- Z := X mod P |
    filter( Z, P, X, Xs, Ys ).

filter( Z, P, X, Xs1, Ys0 ) :- Z =\= 0 | Ys0 = [X|Ys1],
    filter( Xs1, P, Ys1 ).
filter( 0, P, _, Xs1, Ys0 ) :- true |
    filter( Xs1, P, Ys0 ).
```

```
pascal( N, Row ) :- N>0 |
    makeRows( 1, N, [], Row ).

makeRows( K, N, HalfRow, Row ) :- K =:= N,  Odd := N mod 2 |
    fillout( Odd, [[1,0]|HalfRow], Row ).
makeRows( K, N, HalfRowK, Row ) :-
    K < N,
    K1 := K+1,
    Odd := K mod 2 |
    makeRow( [[1,0]|HalfRowK], Odd, HalfRowK1 ),
    makeRows( K1, N, HalfRowK1, Row ).

makeRow( [_], 0, A ) :- true | A=[].
makeRow( [X], 1, A ) :- true | A=[S],  big_plus( S, X, X ).
makeRow( [X1, X2|Xs], Odd, A ) :- true |
    A=[S|Ss],
    big_plus( S, X1, X2 ),
    makeRow( [X2|Xs], Odd, Ss ).

fillout( 0, HalfRow, Row ) :- true |
    rev( HalfRow, [_|Rev] ),
    append( HalfRow, Rev, Row ).
fillout( 1, HalfRow, Row ) :- true |
    rev( HalfRow, Rev ),
    append( HalfRow, Rev, Row ).

rev( X, Y ) :- true | rev( X, [], Y ).

rev( [], Y, Z ) :- true | Z=Y.
rev( [A|X], Y, Z ) :- true | rev( X, [A|Y], Z ).

big_plus( A, X, Y ) :- true | bigp( X, Y, A, 0 ).

bigp( [X|Xs], [Y|Ys], A, C ) :- TmpA := X+Y+C |
    bigp1( Xs, Ys, A, TmpA ).
bigp( [], [Y|Ys], A, C ) :- TmpA := Y+C |
    bigp1( [], Ys, A, TmpA ).
bigp( [X|Xs], [], A, C ) :- TmpA := X+C |
    bigp1( Xs, [], A, TmpA ).
bigp( [], [], A, 0 ) :- true | A=[].
bigp( [], [], A, C ) :- C>0 | A=[C].

bigp1( Xs, Ys, As, TmpA ) :- TmpA >= 100000,
    D := TmpA - 100000 |
    C = 1,
    As = [D|A2],
    bigp( Xs, Ys, A2, C ).
bigp1( Xs, Ys, As, TmpA ) :- TmpA < 100000 |
    C = 0,
    As = [TmpA|A2],
    bigp( Xs, Ys, A2, C ).
```

Performance Evaluation of the Monaco Compiler and Runtime Kernel

E. Tick and C. Banerjee

CIS-TR-92-21
November 1992

Abstract

Monaco is a native-code, shared-memory multiprocessor testbed implementation of flat committed-choice languages such as Flat Guarded Horn Clauses. A reduced abstract-machine instruction set facilitates decision-graph code generation and traditional optimizations based on dataflow analysis. An inexpensive system-call interface is provided to lower the overhead of memory allocation and procedure invocation. In this paper we describe the compilation process and present empirical measurements characterizing performance gains enabled by various compilation and runtime kernel construction techniques.

This paper has been submitted to the *International Conference on Logic Programming*, Budapest, June 1993.

Department of Computer and Information Science
University of Oregon

1 Introduction

High-performance logic programming language implementations require advanced compilation techniques. Specialized hardware may reduce execution times compared to conventional microprocessors, but the *utility* of these optimizations can be low. Compiler optimizations have higher utility because the compiler is usually customized for a particular language family, whereas hardware must be generalized to all languages of practical interest. After initial forays building logic programming machines (e.g., [8, 25, 24]) based on the Warren Abstract Machine (WAM) [2], advanced compilers were built to better exploit conventional microprocessors [39, 34, 17].

Specifically with respect to the development of committed-choice logic programming language compilers, a combination of techniques are required to achieve high performance on conventional hardware. Compilation can be viewed as a front-end code generator and a back-end code optimizer. For both sides, abstract machines based on the WAM, e.g., KL1 [16] and Parlog [5], must be discarded in favor of reduced instruction sets that allow fine-grain manipulations. For example, Kliger and Shapiro's decision-graph method for code generation [18] permits fast clause selection as an alternative to conventional indexing methods [2]. Data-flow analysis in the back-end facilitates standard optimizations, such as dereferencing and type-checking elimination, that are impossible in WAM-like instruction sets.

This paper describes the Monaco compiler and runtime system. The compiler translates flat committed-choice logic programs into an abstract machine suitable for execution on shared-memory multiprocessor hosts, targeted to RISC architectures in particular. Our current implementation accepts FGHC programs and produces native Intel 80386 code [15] for the Sequent Symmetry [26]. Monaco's main purpose is to be a testbed for the advanced optimization techniques such as type inferencing [3], mode analysis [38], sequentialization [23], and local memory reuse [33]. Furthermore, the motivation for developing the compiler was to more accurately characterize the parallel execution behavior of concurrent logic programs by avoiding inefficient emulation, a problem in former studies [35].

The paper is organized as follows. Section 2 briefly reviews literature in the area of compilation of concurrent languages. Section 3 outlines the Monaco instruction set and storage model. The compiler is described in Section 4 and the runtime system in Section 5. Performance characteristics are presented in Section 6. Conclusions and future work are given in Section 7.

2 Literature Review

Emulation-based, real parallel, shared-memory implementations of committed-choice languages include Panda and JAM. Panda [29, 36] implements a subset of Flat Guarded Horn Clauses (FGHC) [37], sharing the PDSS compiler [4], and utilizing an abstract machine instruction set similar to WAM [16]. Jim's Abstract Machine (JAM) [5] is a Parlog emulator including support for Or-Parallel execution of *deep* guards. This introduces additional overheads for process management, since a process activation tree must be maintained. We borrowed a great deal from these systems for Monaco's runtime system, since our objective was to examine compilation issues. Neither of these systems uses optimizing compilers.

Strand [10] is also emulation-based and real parallel, although mapped to an intermediate distributed-memory model, allowing portability to alternative hosts. The language is flat Parlog with *assignment*, essentially equivalent to fully-moded FGHC [38]. The Strand compiler is a commercial product, and thus detailed information is scarce. Strand has a performance advantage when exploiting assignment, but also a potential disadvantage in the overheads incurred when mapping its distributed-memory model onto a shared-memory host. We reconfirm in Section 6 the performance ordering Strand > JAM > Panda. We also demonstrate Monaco's performance relationship to these three systems.

Further restricted from Strand is Janus [28] where the programmer must declare a single producer and consumer for a stream. An experimental Janus-to-C compiler `jc` has been recently developed for uniprocessors [11]. jc has a main advantage over Monaco in that the backend C compiler can do much better register allocation on the host measured in this paper (an 80386-based Symmetry). Like jc, a recent FCP compiler by Kliger is targeted to uniprocessors [17], although it is emulated at present. Monaco's decision-graph compilation method is borrowed from [18] — Monaco extends this locally with dataflow analysis, whereas [17] extends this globally with abstract interpretation to derive procedure bodies optimized for different call sites. Over an extensive set of benchmarks, Kliger reports speedups of 3.2 due to decision graphs over standard indexing, 1.2 due to his global optimizations, and 5.2 due to 68000 native-code compilation [17]. These results encourage us that Monaco is balanced in the sense of putting our effort where the highest payoffs occur.

Finally, we mention RISC-based microprocessor architectures for committed-choice languages: Carmel [12], PIM/i [30], PIM/p [21], and UNIRED-II [32]. These implementations are akin to Monaco, however, they are experiments in specialized hardware, not compilation technology. For example, the macro-call facility in PIM/p is similar in purpose to Monaco's millicode call (Section 4.4), but the hardware implementation is

far more efficient than our interface to C.

3 Monaco Abstract Machine

Committed-choice languages [31] differ from Prolog in several ways, leading to efficient abstract machine definitions that differ from the WAM. First, they have a process-based computation model that does not support backtracking. A computation consists of *reducing* goals (fine-grain tasks or processes), until no unreduced goals remain, in which case the computation succeeds. This implies that fast selection of a committing clause is paramount, as engendered by decision-graph code generation [18]. Second, unification is constrained to be either passive or active. Active unification is more costly than in Prolog because locking is needed to ensure atomic variable binding. Furthermore, to avoid creating circular structures and potential deadlock during multiple unifications of shared variables, we need to create a binding protocol. Third, there is a wide gap in memory-usage efficiency in concurrent and sequential languages. Even parallel Prolog can exploit stacks because of the inherently sequential nature of its threads (e.g., [22, 13]). In committed-choice languages, without sophisticated compiler analysis (e.g., [23]) all goals are potentially concurrent, therefore goal allocation is usually done on a heap. Also, data structures in logic programs are dynamically created and modified, requiring heap storage, although in Prolog, backtracking can naturally reclaim portions of the heap. Overall, the required memory bandwidth of committed-choice languages is significantly greater than that of sequential logic languages. Fourth, process management, i.e., enqueueing, suspending, and resuming operations, must be reduced in frequency and overhead.

The Monaco abstract machine supports seven programmer-visible tagged data types: integers, immediates (symbolic atoms), reference pointers, logical variables, hooked variables, list pairs, and vectors. Abstractly, each of the first five simple objects occupies a machine *cell* consisting of a data field, tag, and lock. As in WAM, a compound object is represented as a pointer to a group of cells containing the list pair or vector. On Symmetry, we implement a cell in eight bytes, with a low-order 3-bit tag in the first word, and the second word currently devoted only to the DYNIX [26] lock. This scheme is similar to that used in Panda, chosen for its simplicity and extensibility.

A logical variable cell is an uninstantiated location, with the data field pointing to the cell itself (as in WAM). When a variable becomes instantiated, its type changes to that of the value that the cell is instantiated with. A *hooked* variable cell is an uninstantiated location on which some process(es) are blocked. The data field of the cell points to a list of suspended processes, as in Panda and JAM. If a hooked cell is

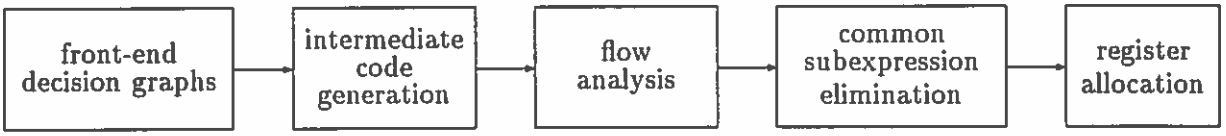

Figure 1: Monaco Compiler Organization (Main Phases Shown)

bound, the processes hooked on the cell must be resumed. Tradeoffs concerning these design decisions are well documented [2, 5].

The Monaco abstract machine is a load-store architecture to facilitate mapping to RISC hosts. Other load-store instruction sets for logic programming languages exist (e.g., [14, 39, 12, 30, 21, 32]) although they have been primarily designed for specialized hardware. Our instruction set is summarized in Table 1. In the table, `Rs`, `Rs1`, and `Rs2` denote source registers, `Rd` denotes a destination register, `N` denotes an integer constant, and `F/A` is the name of an executable procedure.

A majority of the instructions are lightweight and can easily be translated into small sequences of instructions on the host. Most predicate and arithmetic instructions fall into this category. At the other extreme, some instructions are sufficiently complex that not much can be gained by translating them into native code. These are implemented by an interface to the runtime system, called a *millicode call* (see Section 4.4). The middle ground is covered by the data manipulators and constructors. We currently implement these in native code, at some expense in code size, to minimize system-call frequency.

Both active and passive unification are sufficiently complex to warrant implementation in millicode. Full passive unification in the head or guard is only needed in the rare case of shared variables, thus we do not review this algorithm here (see Duvvuru [9]). Active unification commonly occurs in the body, implemented here with the `unify` instruction. In a crop of recent languages, the full power of flat committed-choice languages has been limited, allowing compile-time customization of almost all unification operations. We are particularly interested in moded FGHC (equivalent to Strand), and have built an associated mode analyzer [23]. With derived mode information, we can compile unifications into cheaper assignments, an optimization we examine in Section 6.2. We discover, however, that without more powerful analysis, the general overloading of logic variables for synchronization causes inefficiencies even for assignment.

4 Compilation

The Monaco compiler is summarized in Figure 1. The pipeline follows a traditional

Instruction	# 80386 instr‡	Semantics
arithmetic and predicates		
`iadd(Rs1,Rs2,Rd)`	3	integer arithmetic (`isub`, `idiv`, `imod`, `imul`)
`iand(Rs1,Rs2,Rd)`	3	bitwise arithmetic (`ineg`, `inot`, `ior`, `ixor`)
`ishl(Rs1,Rs2,Rd)`	3	shift left (`ishra`, `ishrl`)
`ieq(Rs1,Rs2,Rd)`	14?	integer comparison (`ige`, `igt`, `ile`, `ilt`, `ineq`)
`isatom(Rs,Rd)`	9?	type comparison (`isbound`, `ishooked`, `isint`, `islist`, `isnil`, `isref`, `isstruct`, `isunbound`)
`eq(Rs1,Rs2,Rd)`	5	comparison (`neq`)
control		
`br(a,L)`	1	jump to L
`br(Cond,Rs,L)`	2	if (R_s = Cond) then branch to L
`switch(Table,Rs)`	2	jump indirect (on tag(R_s)) through Table
`proceed`	6	complete process
`execute(F/A)`	1	execute process F/A
process management		
`punify(Rs1,Rs2,Rd)`	7†	passive unify
`unify(Rs1,Rs2)`	6†	active unify
`enqueue(Rs)`	5†	push goal on ready queue
`push(Rs)`	8	push address onto suspension stack
`suspend(F/A)`	10†	suspend process F/A
data constructors		
`alloc(N,Rd)`	6	allocate heap by N cells
`ref(Rs,N,Rd)`	3	$R_d := R_s + N$
`initvar(Rs)`	4	mem(R_s) := unbound variable
`initlist(Rs1,Rs2,Rd)`	5	mem(R_d) := pair(R_{s1}, R_{s2}); $R_d :=$ pair ptr
`initstruct(N,Rd)`	11	mem(R_d) := vector of size N; $R_d :=$ vec ptr
`initgoal(N,F/A,Rd)`	12	mem(R_d) := goal record; $R_d :=$ vec ptr
`mkunbound(Rd)`	9	$R_d :=$ ptr to unbound variable
`mkconst(C,Rd)`	1	$R_d := C$ (constant)
`mklist(Rs1,Rs2,Rd)`	11	$R_d :=$ ptr to pair(R_{s1}, R_{s2})
`mkstruct(N,Rd)`	10	$R_d :=$ ptr to vector of size N
`mkgoal(N,F/A,Rd)`	11	$R_d :=$ ptr to goal record
data manipulators		
`move(Rs,Rd)`	2	$R_d := R_s$
`deref(Rs,Rd)`	5	$R_d :=$ dereference of R_s
`car(Rs,Rd)`	3	$R_d :=$ head of R_s
`cdr(Rs,Rd)`	3	$R_d :=$ tail of R_s
`sref(Rs,N,Rd)`	3	$R_d :=$ value of N^{th} slot in vector R_s
`sset(Rs,N,Rd)`	3	N^{th} slot in vector $R_d := R_s$
`ssize(Rs,Rd)`	11	$R_d :=$ size of vector R_s

‡ most frequent path, unless otherwise specified.
? longest path.
† millicode call and return only.

Table 1: Monaco Abstract Instruction Set (Committed-Choice Subset)

organization, where the input is a source program and the output is an equivalent program in the abstract machine instruction set. The front-end parses the program and generates decision graphs with Kliger's algorithm [18]. Alternatively, "don't know" procedures (determinacy testing trees) in languages such as Pandora and Andorra can be generated [20]. These graphs and trees are fed to a code-generation phase which produces rudimentary abstract machine code (assuming an infinite register set).

Flow analysis proceeds by decomposing the program into a flow graph of basic blocks. Type and dereferencing information is propagated through the flow graph in the subsequent phase. At this point, macro-instructions are resolved, redundant computations are recognized and eliminated. Deadcode elimination is a minor pass not show in the figure. Register allocation is performed as the final flow graph optimization. The output from the register allocator is an abstract machine program instantiated with abstract register identifiers.

A series of minor phases (not shown) are then required to perform jump-to-jump short circuiting, dead block removal, branch removal, code flattening, and peephole optimization (in that order). The final output is ready for assembly. Our current assembler translates the Monaco program into 80386 assembly code, and invokes the Symmetry assembler for construction of an object file. The object file can then be linked to the precompiled runtime system, producing an executable. We leverage the Symmetry assembler, requiring this somewhat clumsy linkage protocol, because it is portable (for another host, we target our assembler to the host's assembly language, and ignore intricacies of object image formats).

4.1 Front-End: Decision-Graph Generation

Decision graphs have been shown by Kliger [18] to be effective means of rapidly determining which clause within a procedure can commit. Furthermore, these graphs are space-linear in the number of clauses. We use Kliger's algorithm in our front-end: the specifics are discussed in [19]. For each procedure, a cannonical normalized form is produced. The graph is then generated, where each node is a test (e.g., $X > Y - 3$) and edges are valuations of a test (e.g., yes/no or case values). The leaves of the graph are clause bodies, where we lump *tell* operations with the body. The backend will transform a graph into a sequence of triples suitable for optimization.

The key point here is to compute *residuals* which are clause sets that satisfy a guard (ask) test. Satisfaction requires proving implications between clause constraints and the guard. In general, such proofs are difficult since the domain is unspecified. Furthermore, since multiple residuals may be needed per graph node, this computation is critical to front-end efficiency. Our solution to these concerns is to safely approximate

```
check( X, C, _, NCs, Cs, L, S0, S1 ) :- X = [] |
    append( NCs, Cs, Ps ),
    queen( Ps, [], [C|L], S0, S1 ).
check( X, C, D, _, _, _, S0, S1 ) :-
    X = [P|_], P-C =:= D |
    S0 = S1.
check( X, C, D, _, _, _, S0, S1 ) :-
    X = [P|_], C-P =:= D |
    S0 = S1.
check( X, C, D, NCs, Cs, L, S0, S1 ) :-
    X = [P|Ps], P-C =\= D, C-P =\= D,  D1 := D+1 |
    check( Ps, C, D1, NCs, Cs, L, S0, S1 ).
```

Figure 2: Normalized Procedure `check/8` from *queens*

the proofs by table lookup. For example, the constraint $X > I$ implies the constraint $X > J$ if constant integers $I > J$, for variable X. A set of these relations has been found to be quite effective in allowing optimal graph generation. Complex inferences cannot be made (such as transitivity); however, we have not seen such complexity in typical programs.

Consider the `check/8` procedure in the *queens* program, listed in Figure 2. We chose this example to illustrate both the strengths and weaknesses of decision graph compilation. The first step of processing is to create a normalized canonical form from the source program. This entails flattening the head, pulling all complex terms out into guard "ask" unifications. Furthermore, integer type checks are inserted in the guard for all variables involved in arithmetic expressions, e.g., `integer(P)` is included in clauses 3–5. This allows the indexer to choose type checking early in the decision-graph generation phase, resulting in an efficient graph, shown in Figure 3.

The important points to note in the graph are the indexing choice of switching on the first argument, and the placement of integer type checks for `P`, `C`, and `D`. Switching on the first argument is in fact not optimal when considering *call forwarding*, i.e., shorting callers around operations (such as integer checks in this case) that are known *a priori* from flow analysis. We would like the integer checks as high in the graph as possible. One idea is to force the checks up by inserting additional `integer` guards *in the end-of-recursion case*, although that is neither a satisfying nor automatic solution.

Another flaw in the decision graph is that the path to clause two (node 2) requires *three* arithmetic inequalities. Ideally, node (1) should commit to clause two. The problem lies in the power of our inferencing mechanism computing residuals. A clause is not placed in the residual of a branch test unless a guard in that clause *implies* the test (thus retaining space linearity). Actually, $P - C = D$ *does* imply $C - P \neq D$, but we have refrained from adding such inferencing smarts to the compiler. Our inferences

```
switch(X,
  [case([],commit(clause 1))
   case('.'/2,
      ask(integer(P),
          yes(ask(integer(C),
                  yes(ask(D),
                          yes(ask((C-P =\= D),
                                  yes(ask((P-C =\= D),
                                          yes(commit(clause 4))
                                          no(go(7))                    (1)
                                        7:other(go(6)))),
                                  no(commit(clause 3)),
                                6:other(
                                    ask((P-C =:= D),
                                        yes(commit(clause 2)),         (2)
                                        no(go(8)),
                                      8:other(go(5))))))),
                          no(go(5)),
                        5:other(go(4)))),
                  no(go(4)),
                4:other(go(3)))),
          no(go(3)),
        3:other(go(2)))),
  2:default(go(1))]
1:suspend
```

Figure 3: Stylized Decision Graph for `check/8`

are purely table driven, and although we could insert many more table entries for such ad hoc cases, a general prover would be best. A final point: the rather circuitous routes to suspension are entirely collapsed by dataflow analysis and jump-chain shorting in subsequent phases of the compilation.

4.2 Code Generation

Code generation translates the decision graphs into intermediate Monaco instructions. The translation algorithms are straightforward, perhaps naive concerning arithmetic expressions. Key to code generation is a "one register — one value" invariant that facilitates all later phases, e.g., a register can be defined only once within a procedure. Perhaps the trickiest part of code generation is the construction of switch-on-tag statements, which are composed of eight binary switch instructions, each in its own basic block. This allows us to standardize basic blocks in a binary tree. Body generation follows the standard style (e.g., [16]) of enqueueing all body goals but the first, which is executed immediately.

Subsequent dataflow analysis proceeds from the Monaco code, driven by a table describing each instruction's operand uses and definitions, as described in the next

section. In retrospect, we found that certain analyses, such as call forwarding [6], is best done on the decision graph, not the generated code. Although dataflow information must be derived earlier to do this, it is much easier to rearrange portions of the flow graph, with no concern for register bindings.

4.3 Flow-Analysis Optimizations

Data-flow analysis is fundamental to most of the compiler optimizations, to the point where the preliminary code is particularly naive and *requires* flow analysis to clean it up. We took this approach to keep the compiler modular, although it impacts compile time. The basis for flow analysis is the construction of a flow graph of basic blocks from the preliminary code. A standard construction algorithm is used [1] with the note that Monaco instructions such as `execute`, `proceed`, and `suspend` represent control transfers, and thus terminate blocks.

Note that the program is analyzed locally, i.e., on a procedure-by-procedure basis. A flow graph, created for a procedure, is topologically sorted to ensure that all ancestors of a child block are analyzed before the child is analyzed. Type information is then collected and propagated from the root, taking the set intersection (as the least upper bound) of information arriving to a child node from its ancestors. The information is essentially an association list matching pseudo-registers and their abstract contents. For example, suppose we know `R3 = car(deref(R1))`, and we encounter a Monaco instruction `deref R3,R4`, then we derive the new information: `R4 <- deref(car(deref(R1)))`. A common association entry is `R6 <- R5`, created from the instruction `move R5,R6`.

Code within the blocks is rewritten on the fly during analysis to share common subexpressions. The most common case of this is shorting moves, e.g., `move R5,R6` followed by `isint R6,R7`, will rewrite the latter instruction to be `isint R5,R7`. Later deadcode elimination is of course required here. Branch conditions are also propagated throughout the flow graph. We must be careful to distinguish the taken condition from the not-taken condition. For example, entering some block in the graph, we may know that `integer(deref(car(deref(R1))))` must be true, by consideration of the flow to that point. This allows branch shorting, e.g., `br nz,R8`, which branches if `R8` is not zero, can be combined with flow information `R8 <- isint(deref(car(deref(R1))))` and the previous branch information, to derive that the branch is always taken, and so it will be rewritten as an unconditional jump.

Deadcode elimination and register allocation require further flow graph analysis. Live-range analysis [1] is performed on the graph, producing the information of which registers are live within each basic block. We chose to collect this information at

block granularity rather than instruction granularity to reduce compile time. Large clause bodies can have relatively large basic blocks, reducing the effectiveness of this technique. For this reason we artificially split such blocks at each body goal. In general, this heuristic is sufficient to retain accuracy, as discussed in Section 6.

If a value is never used, then its live range will be empty. Hence, deadcode elimination is performed by removing instructions containing values with empty live ranges. Give the previous analysis, this phase is trivial. Next, we allocate registers locally to each procedure.[1] The local allocation method used is based on the liveness of the registers, and is performed on a basis-block granularity to match live-range analysis. The most live name (i.e., having the most basic blocks over its life) is allocated first, and so on. The algorithm is non-backtracking, so lack of an available register for the next most frequent name requires generation of spill code. The spill is allocated to a vector local to the procedure.

The goodness of such a naive scheme relies on the accuracy of flow analysis. Allocating on a block basis can lead to frequent spill code, although this is alleviated by splitting the body along individual goals. Still, a goal requiring the evaluation and loading of many actuals is the most likely to cause spills. Spilling can be reduced by artificially splitting the blocks into finer grains (in the limit: instructions). Because we were targeting our initial experiments to an 80386 backend with so few registers available to the program, it became hopeless to avoid frequent spilling. Instead we generate memory accesses to a pseudo-register array. This array is small enough to easily fit in cache, yet large enough to avoid spilling. This stop-gap measure allows us to refine the compiler for eventual porting to a host with sufficient registers.

Our allocator averages 7.3 registers for the 37 benchmark procedures considered in Section 6. Trivial procedures ranged from 2–7 registers, whereas more complex procedures ranged from 11–17 registers. Future work includes implementing a global allocator based on cooperation across procedure call boundaries.

Final phases include shorting of all jump chains, which leads to dead basic blocks that are removed. The flow graph is then flattened and a peephole optimizer filters the code stream. Currently the peepholer is limited: its primary function is to attempt common subexpression elimination (CSE) of spill sequences. This is inherent to the well-known problem of where to allocate registers: before or after CSE. Since we allocate before, we cannot eliminate spill code redundancies. The peepholer cannot make much headway here, and we plan to explore a split register allocator, as in gcc, to better solve the problem.

[1] Global register allocation in concurrent languages such as FGHC first requires sequentialization of threads, which we are pursuing[23].

We translate the Monaco code into 80386 Symmetry assembly code in a final assembler module. The 80386 code is reassembled with the Symmetry assembler, producing an linkable object file. The object is linked directly to the runtime system object, forming a executable file (customized to this particular source program). This method is quick and simple — the system object need only be linked, not recompiled. Only objects are saved — executables are deleted after use. This is sufficient for our use as an experimental testbed.

4.4 Example

The Monaco code for the quicksort procedure written in FGHC, is shown in Figure 4 (`part/4` not shown). The precise semantics of the instruction set are not necessary to follow the example. Before entry to a procedure, the input arguments are loaded in to the abstract machine registers numbered from zero. The entry point to this procedure is labeled `qsort/3`. Arguments are dereferenced before they are tested (1). Quicksort dispatches on its first (input) argument in a switch on tag instruction (2). If unbound, control flows to L1, where the argument is pushed on the suspension stack and suspension occurs (3–4). If an immediate or vector, control flows to L2 where the suspension routine is called with no pushed arguments (4), resulting in failure. If nil, control flows to L3, where the argument is compared to nil and either fails or commits to the first clause (5–9). If the first argument is a list, the second clause commits at L4 (10–35). The `alloc` instruction (10) allocates 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 `part/4` procedure, after setting up its arguments in appropriate registers, exploiting last-call optimization. We will comment more about how this code is generated in the next section.

There are several interesting points about how code is generated from the decision graph. These characteristics are analyzed empirically in Section 6.

- Originally binary branches were favored over switches, because we planned Monaco for a RISC backend. However, the 80386, based on a condition code register is not conducive to branching on conditions stored in registers. Therefore we introduced the `switch` instruction, assembled into an indirect jump-on-tag table.

- Structures are created on the heap by allocating space for the entire clause body, and then filling it in for each structure (including goal records). For each structure argument, a reference is built (with `ref`) pointing to the heap location where it will be constructed, and the storage is initialized (e.g., `initvar` and `initgoal`).

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

qsort/3:  deref    r0,r4          % r4 := deref(Argument 1)          1
          switch   [L1,L4,L3,L2,L2,L1,L1,L1],r4                      2
L1:       push     r0             % No. Push Argument 1. Suspend.    3
L2:       suspend  qsort/3        % Suspend                          4
L3:       mkconst  [],r3          % Argument 1 is []?                5
          eq       r4,r3,r5                                          6
          br       z,r5,L2        % No, fail.                        7
          unify    r1,r2          % Yes. Rest = Ans                  8
          proceed                                                    9
L4:       alloc    15,r5          % Allocate 15 cells on heap.       10
          move     r5,r6                                             11
          initgoal qsort/3,r6     % r6 -> goal qsort/3               12
          ref      r5,5,r3        % r3 := L                          13
          initvar  r3                                                14
          sset     r3,1,r6        % 1(r6) := L                       15
          ref      r5,6,r8        % r8 := Y1                         16
          initvar  r8                                                17
          sset     r8,2,r6        % 2(r6) := Y1                      18
          sset     r2,3,r6        % 3(r6) := T                       19
          enqueue  r6             % enqueue qsort(L,Y1,T)            20
          ref      r5,7,r9                                           21
          initgoal qsort/3,r9     % r9 -> goal qsort/3               22
          ref      r5,12,r2       % r2 := S                          23
          initvar  r2                                                24
          sset     r2,1,r9        % 1(r9) := S                       25
          sset     r1,2,r9        % 2(r9) := Y                       26
          car      r4,r1          % r1 := X                          27
          ref      r5,13,r12                                         28
          initlist r1,r8,r12      % r12 := [X|Y1]                    29
          sset     r12,3,r9       % 3(r9) := [X|Y1]                  30
          enqueue  r9             % enqueue qsort(S,Y,[X|Y1])        31
          cdr      r4,r0          % r0 := R                          32
          execute  part/4         % call part(R,X,S,L)               33
```

Figure 4: Compiled Quicksort Procedure in Monaco

Then the structure argument position is bound to the reference (with `sset`). This implies that variables reside outside of structures, which will ease future optimizations concerning local memory reuse [33].

- The final Monaco code shown in Figure 4 is the result of extensive dataflow analysis and optimization. With no optimizations the procedure has 43 Monaco instructions, an increase of 30%. The primary savings in code size, and correspondingly execution time, comes from CSE and tight control.

- The compiler was purposely kept as elegant and modular as possible, with a minimum number of stages, and therefore less than optimal code is produced. For example, we chose to do register allocation *after* CSE. Thus if we have register spilling, the spill sequences will not be simplified. We also chose to abide by a code generation policy wherein every pseudo register is defined at most once per procedure. Given this policy, which facilitates dataflow analysis, we need to assign *different* pseudo registers to passed parameters and tail-recursive arguments. Those registers need to be coerced during register allocation. The result is poor allocation in the basic block containing the tail recursion. Instead of more dataflow analysis to solve this [27], a patch was added to scan the block in reverse, shorting out register-to-register moves. This produces reasonable, but not excellent, code.

The macro assembler is the penultimate stage in the compilation process. It translates a Monaco assembly program into an 80386 assembly program, for final translation by the DYNIX assembler *as*. Figure 5 gives two examples of macro expansion of Monaco into 80386 assembly. Instructions `unify`, `punify`, `enqueue`, `suspend`, and `execute` are sufficiently complex to warrant implementation within the runtime system as millicode. As shown for `punify`, a millicode call is accomplished by means of an indirect jump table, with parameters passed as in 'C'.

5 Monaco Runtime System

In this section we give a brief overview of the runtime kernel to put the subsequent performance analysis in perspective. For a detailed discussion, see Duvvuru [9].

Concurrent logic languages, due to the nature of the fine-grain tasks, single-assignment variables, and heap-based memory management, require extensive runtime support. The main objectives of the runtime system are to support fast process management, memory allocation, and runtime system calls, as described in this section. Like most WAM-based systems (e.g., [22, 29, 5]), the Monaco kernel is based on "task farming:"

```
        movl    Rs*4(REGS),TMP0     # deref(Rs, Rd)
        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

                                    # punify(Rs1, Rs2, Rd)
        pushl   Rs2*4(REGS)         # push Rs2 on stack
        pushl   Rs1*4(REGS)         # push Rs1 on stack
        pushl   $LM_RET             # push return address
        jmp     *PUNIFY_OFF(REGS)   # indirect call to punify
LM_RET: addl    $8,%esp             # readjust stack
        movl    TMP0,Rd*4(REGS)     # Rd := punify(Rs1,Rs2)
```

Figure 5: Examples of 80386 Macro Expansion on Symmetry (TMP0, TMP1 and REGS are machine register aliases)

a set of *worker* processes that run as regular DYNIX processes on all the processors. A worker process corresponds to one Monaco abstract machine processor and operates in a loop: looking for reducible goals in the pool, finding one and reducing it, adding the body goals of that clause to the pool, and continuing to look for more work.

To reduce synchronization, the goal pool is split among the workers — each has a private goal queue, implemented as a list. Thus we both enqueue and dequeue goals from the queue head, resulting in pseudo depth-first execution (not pure because idle workers steal goals from busy worker's queues). Depth-first evaluation has been shown to be relatively efficient [29], so we adopted it here. We also adopted the JAM scheduling strategy of a ring: an idle worker examines the private queues of successive clockwise neighbors until a goal is found. This scheme is efficient because the idle worker responsible for finding work, affecting busy workers only when synchronizing on the queue.[2] This scheme works effectively and produces reasonable load balancing, as shown in JAM and here (see Section 6). Our current research emphasis is in an efficient instruction set and compilation procedure, so we have adopted standard techniques for process management. Suspension and resumption is done with logical variables hooked to suspended goals via suspension slips, as in JAM Parlog [5] and KL1 [16] implementations.

Most goals allocate heap memory for creating data and body goal structures. We

[2] JAM reduces synchronization by stealing from the tail of the queue. We did not implement this optimization because we plan to use priority queues exploiting granularity estimations [40] in future versions of Monaco.

Program	Monaco	Strand	JAM	Panda	PDSS	Janus
hanoi(14)	1.0	1.3	1.7	2.1	2.1	0.27
nrev(1000)	1.0	1.8	2.0	3.3	2.9	0.20
qsort(1000)	1.0	1.9	2.4	5.2	2.7	0.60
queens(10)	1.0	0.59	3.2	4.6	4.2	0.91
primes(5000)	1.0	3.0	3.7	5.8	3.7	0.48
pascal(200)	1.0	2.4	2.3	4.4	2.6	1.85†
geo. mean	1.0	1.6	2.5	4.0	3.0	0.54

† not optimized due to compiler bug.

Table 2: Uniprocessor Execution Time Normalized to Monaco on Sequent Symmetry (Full Optimizations)

reduce the frequency of allocation by aggregating all heap requests within a clause into a single millicode call. The Monaco heap is split into equal-sized slices for each worker, avoiding synchronization. Each heap is initialized with two range pointers delimiting the unallocated area. All heap allocations, approximately one per reduction, manipulate and check these pointers accordingly. The overflow check is sufficiently frequent that we chose to generate it in native code. This naive heap implementation is sufficient for our initial evaluation of compilation techniques. The current version of Monaco does not yet implement garbage collection (and so collection overheads are not accounted for in the empirical analysis given in Section 6).

6 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 Strand (Buckingham) [10], JAM V1.4 [5], PDSS V2.52.19 [4], Panda (using PDSS V0.8 compiler!) [29], and Janus [11]. For multiprocessor execution, comparison is made only with JAM because it is the fastest of the parallel systems with comparative scheduling. Empirical evaluation was conducted on a Sequent Symmetry S81 system with 16 MHz Intel 80386 microprocessors. Execution times (in milliseconds) are calculated as the elapsed time between starting the computation, until all the processors become idle. All measurements presented are the minimum of several runs.

Table 2 shows uniprocessor (one PE) execution times, normalized to Monaco, for a benchmark suite (listed in the Appendix). Monaco is compiled with full optimiza-

tions, but without mode analysis (this primarily affects *queens* as mentioned below). The measurements are only an approximate comparison: the systems offer different facilities (e.g., language capabilities, types of garbage collectors) that make a fair comparison difficult. For example, PDSS and Janus are sequential systems without any multiprocessor overheads. In any case, these measurements satisfy us that the Monaco has slightly better performance than alternative multiprocessor systems.

We still have a far way to improve (10%–500%) to achieve Janus speeds (of course, we can make this up with a multiprocessor). With respect to Strand, the fastest competition publicly available, we are running from 40% slower to three times faster. We believe the 40% gap in *queens* is due to 1) a non-optimal decision graph, as described in Section 4.1; 2) our somewhat naive code generation of arithmetic expressions (no static type inferencing is done); and 3) no exploitation of mode information (to strength reduce variable-to-variable unifies, which comes naturally in Stand). We discuss optimizations in Section 6.2 to reduce (2) and (3), resulting in 20% speedup in *queens* which reduces the gap with Strand to −30%.

The multiprocessor execution times of Monaco and JAM are given in Table 3. The speedups, shown in Table 4, are calculated as the ratio of execution times of the same multiprocessor system on one PE to multiple PEs. Thus the metric is biased, but still gives us a first approximation of the utility of exploiting parallelism. Monaco and JAM use 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 as in JAM. Thus we are not surprised that speedups are similar. It is not clear why JAM's execution time *increased* from 4 to 8 PEs on *qsort*, the outlier of the benchmarks. In general we expected the experiment to be biased to JAM which allocates proportionally more heap with increasing numbers of PEs. Again, these are preliminary measurements and we conclude simply that Monaco is successfully exploiting fine-grain parallelism, even though its streamlined abstract machine achieved gains in absolute performance.

6.1 Execution Profiling

We instrumented the Monaco runtime kernel to measure the performance of the frequently-used millicode procedures: `unify`, `enqueue`, and `suspend`. All the measurements were taken with one worker, using tools built from UNIX *profil* and *prof*. Table 5 shows the percentage execution time spent in the user program (Monaco code) and the runtime kernel. The kernel consists of **Monitor** (where the worker steals or dequeues goals and prepares them for execution), **Run** (a stub currently needed to call the Monaco code), **Enq** (millicode for enqueueing goals), **Susp** (millicode for suspending goals), **Stub** (the fast case of unification), **Rest** (the general body of unification), and

Program	1 PE	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco (msec)					
hanoi	4,364	2,339	1,218	722	598	558
nrev	19,153	11,740	6,136	3,394	2,503	2,155
qsort	21,412	12,970	6,621	3,496	2,438	1,954
queens	43,305	30,348	12,413	6,153	4,041	3,082
primes	12,841	7,549	3,788	1,984	1,390	1,117
pascal	8,973	4,981	2,499	1,294	919	724
	JAM (msec)					
hanoi	7,360	3,780	2,260	1,620	1,390	1,290
nrev	38,030	33,550	10,200	5,290	3,570	2,700
qsort	50,900	26,300	14,120	19,150	15,220	13,170
queens	139,590	69,860	34,900	17,490	11,680	8,780
primes	48,120	24,930	12,520	7,300	6,270	5,550
pascal	20,300	10,130	5,140	2,620	1,780	1,360
	JAM/Monaco					
hanoi	1.7	1.6	1.9	2.2	2.3	2.3
nrev	2.0	2.9	1.7	1.6	1.4	1.3
qsort	2.4	2.0	2.1	5.5	6.2	6.7
queens	3.2	2.3	2.8	2.8	2.9	2.8
primes	3.7	3.3	3.3	3.7	4.5	5.0
pascal	2.3	2.0	2.1	2.0	1.9	1.9
geo. mean	2.5	2.3	2.3	2.7	2.8	2.8

Table 3: Multiprocessor Performance of Monaco (Optimized) and JAM

Program	2 PE	4 PE	8 PE	12 PE	16 PE	2 PE	4 PE	8 PE	12 PE	16 PE
	Monaco					**JAM**				
hanoi	1.9	3.6	6.0	7.3	7.8	1.9	3.3	4.5	5.3	5.7
nrev	1.6	3.1	5.6	7.7	8.9	1.1	3.7	7.2	10.7	14.1
qsort	1.7	3.2	6.1	8.8	11.0	1.9	3.6	2.7	3.3	3.9
queens	1.4	3.5	7.0	10.7	14.1	2.0	4.0	8.0	12.0	15.9
primes	1.7	3.4	6.5	9.2	11.5	1.9	3.8	6.6	7.7	8.7
pascal	1.8	3.6	6.9	9.8	12.4	2.0	3.9	7.7	11.4	14.9
geo. mean	1.7	3.4	6.3	8.8	10.7	1.8	3.7	5.7	7.6	9.3

Table 4: Multiprocessor Speedups on Monaco and JAM

			Kernel						
					Goals		Unify		
Program	**User**	**Kernel**	**Monitor**	**Run**	**Enq**	**Susp**	**Stub**	**Rest**	**Misc**
hanoi	47.11	52.89	9.10	16.08	9.47	0.00	11.06	0.00	7.15
nrev	56.24	43.76	0.39	0.33	0.11	0.00	34.61	0.20	8.12
qsort	62.36	37.64	0.39	0.28	0.28	0.00	30.59	0.01	6.11
queens	54.48	45.52	8.84	12.46	8.82	0.00	7.90	0.00	7.47
prime	69.74	30.26	0.53	0.18	0.10	0.00	24.01	0.01	5.34
pascal	54.63	45.37	2.16	2.72	2.61	0.00	29.25	0.10	8.61
arith. mean	57.43	42.57	3.57	5.34	3.57	0.00	22.90	0.05	7.13
geo. mean	57.00	41.94	1.55	1.44	0.94	0.00	20.05	0.00	7.04

Table 5: Percentage Execution Times in Kernel and User Program (Fully Optimized, on One Worker)

miscellaneous. Suspension is negligible in these benchmarks, making **Susp** and **Rest** insignificant. *Hanoi* and *queens* enqueue a large number of goal records and hence the time spent in **Enq**, **Monitor**, and **Run** is significant: on average 32.39% compared to 12.48% over all the programs.

The statistics indicate that a more streamlined thread interface will be beneficial for procedure intensive programs. For example, the **Run** stub is a patch around some compilation difficulties, and in theory is not needed. With a RISC backend, we would avoid loading goal arguments into memory-simulated registers, as we currently do for the 80386, thus reducing **Monitor** time. More far-reaching solutions involve increasing granularity by sequentialization (see next section). A further kernel optimization would be to pull up the unification **Stub** (now averaging 22.9%, the most overhead of any single function) into assembly code.

6.2 Compiler Optimizations

Table 6 shows the performance gains of three optimizations: **CSE**: dataflow analysis enabling common subexpression elimination and subsequent deadcode removal, **switch**: using branch tables for generating switch nodes with the decision graphs, and $X := Y$: the previously described variable-to-variable unification optimization. Each effect is shown as a speedup relative to the previous compilation. In the case of switches, the default method uses a chain of binary branches. In the case of unification, the default method uses a stub that cannot quickly perform variable-to-variable assignment (for fear of deadlock in nonmoded programs). The measurements clearly demonstrate the effectiveness of dataflow analysis, and to a lesser extent, fast branching. Note that

Program	CSE	switch	$X := Y$
hanoi(14)	2.19	5.01	0.65
nrev(1000)	16.06	8.48	−0.50
qsort(1000)	32.21	16.88	−0.22
queens(10)	18.07	12.75	11.66
prime(5000)	27.51	16.61	−0.14
pascal(200)	21.49	12.40	1.44
geo. mean	15.15	11.11	—

Table 6: Percentage Performance Gain with Dataflow Analysis, Branch Tables, and Mode Analysis Optimizations

Experiment	Time (msec)	% Gain
baseline (standard opts)	38,578	
hand register allocation	38,275	0.79
efficient 80386 branching	37,165	2.99
call forwarding	36,136	2.85
total		6.62

Table 7: Additional Optimizations, Measured for Queen(10)

discounting the outlier *hanoi*, CSE accounts for 22% average gain. The assignment optimization benefits only *queens*, the one benchmark that collects multiple solutions in a difference list, and thus shorts the list frequently with variable-to-variable unification. The performance gain/loss for the other benchmarks is insignificant, so we do not calculate a mean.

To experiment with the utility of decision graph code generation, we did not have an alternative frontend that used WAM-style indexing. Instead, we generated decision graphs with a naive indexer. The naive indexer simply chose the next available test for the next node. However, since arithmetic comparisons required that integer types be checked earlier, even the naive indexer chose integer tests first. The benchmark suite was sufficiently simple that the naive indexer produced code that was approximately as fast as the sophisticated indexer. More complex procedures are required to demonstrate a significant difference, e.g., [17].

Three additional optimizations were consider by hand modification of the compiler-generated code for *queens*: optimal register allocation, efficient branching, and efficient calling. The results are given in Table 7. To experiment with the performance loss due to register allocation, we hand-allocated the Monaco registers for *queen*. Compiler

register allocation averaged eight registers per procedure, ranging from 4–15. Hand allocation reduced this to 5.6 registers per procedure, ranging from 2–12 (this included optimistically rearranging instructions to ease allocation). Because these registers are simulated, global allocation has no beneficial effect. The only advantage we can measure is the reduction of move instructions, which as seen has little (0.8%) speedup. Hand allocation of 80386 registers was too intricate to contemplate, even for *queens* with 803 assembly instructions.

The poor match between Monaco's branches and the 80386 was alleviated by a peephole optimizer in the assembler. The problem with this method is that it squashes the branch-condition register, erroneous if that register is live. This optimization is safe for *queens*, allowing us to measure its impact, a 3% speedup.

Call forwarding is an optimization wherein each tail-recursive caller jumps directly to the portion of the callee that is relevant, skipping work that is known to be superfluous. We hand coded this for *queens*, the benchmark program with the most to gain from the optimization, by modifying the intermediate Monaco program. The 2.8% speedup we measured can be calibrated with significantly higher reported gains in [6] by considering that 42% of our execution time is spent in the runtime kernel, not present in Janus. Thus our results might be normalized upwards to 5%.

As a final experiment, we developed an alternative backend for the Monaco compiler which translates FGHC into sequential C code. The main advantages are exploiting the C compiler's 80386 register allocator, and avoiding fine-grain task management overheads. The enabling technology for this is mode and sequentialization analysis [23]. Essentially, fully-moded FGHC programs can be analyzed to determine all path modes nonambiguously, allowing us to impose an ordering on clause body goals that can be sequentially executed, safe from internal deadlock (external producers may still cause suspension). Efficient code can be generated for such programs, illustrated in Table 8 for the *qsort* benchmark. The row labeled "sequential C" is generated by our alternative backend. For instance, the code is 10.5/2.2 = 4.8 times faster than the standard Monaco compiler. The details of the code generation are given in Massey and Tick [23], and are slightly biased because 1) 32-bit cells rather than 64-bit cells (as in Monaco) were used; 2) suspensions due to external sources cannot be handled. In any case, we are encouraged to further develop this backend so that our standard multi-threading can coexist with sequentialized threads.

System	PEs	Problem Size	Time (sec)
PDSS	1	125	1.4
		250	5.2
		500	20.5
Monaco	1	500	10.5
	4	125	0.20
		250	0.75
		500	2.9
	8	500	1.5
sequential C	1	125	0.14
		250	0.58
		500	2.2

System	PEs	Problem Size	Time (sec)
Strand	1	500	10.8
JAM Parlog	1	500	13.1
handcrafted C	1	125	0.10
		250	0.40
		500	1.5

Table 8: Performance of Sequentialized and Parallel QuickSort

7 Conclusions

We have presented the Monaco runtime kernel and compiler, a shared-memory implementation of flat committed-choice languages. In the spirit of Van Roy [39] and Taylor [34], the key design decision was to move from a WAM-based to a lower level intermediate instruction set. This demanded the construction of an optimizing compiler based on local dataflow analysis. Our system is unique in that, 1) it translates concurrent programs onto a parallel execution model; 2) it produces intermediate code targeted for high-performance on RISC hosts; 3) backend generates native code; 4) mode analysis allows the conversion of unifications into assignments in fully-moded programs, and 5) it forms a foundation for global optimizations that can then be accurately measured within a streamlined system. We have presented empirical measurements characterizing the execution profile of the system, demonstrating the utility of the optimizations, and indicating areas for future gains.

Future work involves tuning the kernel and 80386 backend, building a RISC backend, and also a 'C' backend. The latter two promise significantly better register allocation, and producing 'C' will make the compiler portable. We are starting to integrate global optimization techniques within Monaco: sequentialization (underway), local reuse analysis [33], weighted decision graphs [7], and call forwarding [6].

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. L. Hansen designed the original Monaco instruction set. S. Duvvuru completed the first implementation of the Monaco runtime

system. F. Rakoczi implemented the profiling tools for Monaco. The authors thank B. Massey for his kind assistance in this research.

References

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

[2] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[3] M. Bruynooghe, A. Marien, and G. Janssens. The Impact of Abstract Interpretation: and Experiment in Code Generation. In *International Conference on Logic Programming*, pages 33–47. Lisbon, MIT Press, June 1989.

[4] T. Chikayama *et al.* Overview of the Parallel Inference Machine Operating System PIMOS. In *International Conference on Fifth Generation Computer Systems*, pages 230–251, Tokyo, November 1988. ICOT.

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

[6] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan. Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, June 1993. ACM Press. Submitted.

[7] 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.

[8] T. P. Dobry, A. M. Despain, and Y. N. Patt. Performance Studies of a Prolog Machine Architecture. In *International Symposium on Computer Architecture*, pages 180–190. Boston, IEEE Computer Society, December 1985.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[14] 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.

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

[16] 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.

[17] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992. Draft.

[18] 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.

[19] M. Korsloot and E. Tick. Determinacy Testing for Nondeterminate Logic Programming Languages. *ACM Transactions on Programming Languages and Systems.* Submitted, undergoing revision.

[20] M. Korsloot and E. Tick. Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages. In *International Conference on Logic Programming*, pages 457–471. Paris, MIT Press, June 1991.

[21] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa, and K. Hirano. Architecture and Implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, pages 414–424, Tokyo, June 1992. ICOT.

[22] E. Lusk *et al.* The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830, Tokyo, November 1988. ICOT.

[23] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, June 1993. ACM Press. Submitted.

[24] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *International Symposium on Logic Programming*, pages 104–113. San Francisco, IEEE Computer Society, August 1987.

[25] R. Nakazaki *et al.* Design of a High-Speed Prolog Machine (HPM). In *International Symposium on Computer Architecture*, pages 191–197. Boston, IEEE Computer Society, June 1985.

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

[27] F. Rakoczi, K. Peery, and G. Folkestad. Optimizing the Monaco Compiler. Dept. of Computer Science, University of Oregon, June 1992. Unpublished.

[28] 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.

[29] 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.

[30] M. Sato, K. Kato, K. Takeda, and T. Oohara. Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine. Technical Report TR-676, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1991.

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

[32] K. Shimada, H. Koike, and H. Tanaka. UNIRED-II: The High Performance Inference Processor for the Parallel Inference Machine PIE64. In *International Conference on Fifth Generation Computer Systems*, pages 715–722, Tokyo, June 1992. ICOT.

[33] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*. Washington D.C., MIT Press, November 1992.

[34] 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.

[35] E. Tick. A Performance Comparison of AND- and OR-Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 452–470. Lisbon, MIT Press, June 1989.

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

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

[38] 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.

[39] 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.

[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.

Appendix: Benchmark Source Code Listing (FGHC)

```
:- module suite.
:- public nrevgo/1, qsortgo/1, hanoigo/1, queengo/1, primego/1, pascalgo/1.

nrevgo(   N ) :- N > 0 | gen( N, X ), nrev( X, _ ).
qsortgo(  N ) :- N > 0 | gen( N, X ), qsort( X, _ ).
hanoigo(  N ) :- N > 0 | hanoi( N, _ ).
queengo(  N ) :- N > 0 | queen( N, _ ).
primego(  N ) :- N > 0 | prime( N, _ ).
pascalgo( N ) :- N > 0 | pascal( N, _ ).

gen( N, L ) :- true | gen( 0, N, L ).

gen( K, N, L ) :- K >= N | L = [].
gen( K, N, L ) :- K <  N | K1 := K+1, L = [K|Ls], gen( K1, N, Ls ).

nrev( [], Y ) :- true | Y = [].
nrev( [A|X], Y ) :- true | nrev( X, T ), append( T, [A], Y ).
```

```
append( [], X, Y ) :- true | Y = X.
append( [A|B], X, Y ) :- true | Y = [A|Z], append( B, X, Z ).

qsort( X, Y ) :- true | qsort( X, Y, [] ).

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 ).

partition( [X|Xs], A, S, L ) :- A < X | L = [X|L1],
    partition(Xs ,A,S,L1).
partition( [X|Xs], A, S, L ) :- A >= X | S = [X|S1],
    partition( Xs, A, S1, L ).
partition( [], A, S, L ) :- integer(A) | S = [], L = [].

hanoi( N, X ) :- true |
    move( N, left, center, right, Y, [] ),
    count( Y, X ).

move( 0, _, _, _, O1, O2 ) :- true | O1 = O2.
move( N, A, B, C, O1, O4 ) :- N>0, M := N-1 |
    move( M, A, C, B, O1, O2 ),
    O2 = [ A-B | O3 ],
    move( M, C, B, A, O3, O4 ).

count( L,N ) :- true | count( L,0,N ).

count( [], M, N ) :- integer(M) | N = M.
count( [_|Xs], M, N ) :- M1 := M+1 | count( Xs, M1, N ).

queen( N, M ) :- true |
    gen( N, L ),
    queen( L, [], [], A, [] ),
    count( A, M ).

queen( [C|Cs], NCs, L, S0, S2 ) :- true |
    check( L, C, 1, NCs, Cs, L, S0, S1 ),
    queen( Cs, [C|NCs], L, S1, S2 ).
queen( [], [], L, S0, S1 ) :- true | S0 = [L|S1].
queen( [], [_|_], _, S0, S1 ) :- true | S0 = S1.

check( [], C, D, NCs, Cs, L, S0, S1 ) :- integer( C ), integer( D ) |
    append( NCs, Cs, Ps ),
    queen( Ps, [], [C|L], S0, S1 ).
check( [P|_], C, D, _, _, _, S0, S1 ) :- P-C =:= D | S0 = S1.
check( [P|_], C, D, _, _, _, S0, S1 ) :- C-P =:= D | S0 = S1.
check( [P|Ps], C, D, NCs, Cs, L, S0, S1 ) :-
    P-C =\= D, C-P =\= D, D1 := D+1 |
    check( Ps, C, D1, NCs, Cs, L, S0, S1 ).

prime( M,  N ) :- true |
    prime1( M, Ps ),
    count( Ps, N ).
```

```
prime1( Max, Ps ) :- true |
    gen( 2, Max, Ns ),
    sift( Ns, Ps ).

sift( [P|Xs1], Zs0 ) :- true | Zs0 = [P|Zs1],
    filter( Xs1, P, Ys ),
    sift( Ys, Zs1 ).
sift( [], Zs0 ) :- true | Zs0 = [].

filter( [], _, Ys ) :- true | Ys = [].
filter( [X|Xs], P, Ys ) :- Z := X mod P |
    filter( Z, P, X, Xs, Ys ).

filter( Z, P, X, Xs1, Ys0 ) :- Z =\= 0 | Ys0 = [X|Ys1],
    filter( Xs1, P, Ys1 ).
filter( 0, P, _, Xs1, Ys0 ) :- true |
    filter( Xs1, P, Ys0 ).

pascal( N, Row ) :- N>0 |
    makeRows( 1, N, [], Row ).

makeRows( K, N, HalfRow, Row ) :- K =:= N,  Odd := N mod 2 |
    fillout( Odd, [[1,0]|HalfRow], Row ).
makeRows( K, N, HalfRowK, Row ) :-
    K < N,
    K1 := K+1,
    Odd := K mod 2 |
    makeRow( [[1,0]|HalfRowK], Odd, HalfRowK1 ),
    makeRows( K1, N, HalfRowK1, Row ).

makeRow( [_], 0, A ) :- true | A=[].
makeRow( [X], 1, A ) :- true | A=[S],  big_plus( S, X, X ).
makeRow( [X1, X2|Xs], Odd, A ) :- true |
    A=[S|Ss],
    big_plus( S, X1, X2 ),
    makeRow( [X2|Xs], Odd, Ss ).

fillout( 0, HalfRow, Row ) :- true |
    rev( HalfRow, [_|Rev] ),
    append( HalfRow, Rev, Row ).
fillout( 1, HalfRow, Row ) :- true |
    rev( HalfRow, Rev ),
    append( HalfRow, Rev, Row ).

rev( X, Y ) :- true | rev( X, [], Y ).

rev( [], Y, Z ) :- true | Z=Y.
rev( [A|X], Y, Z ) :- true | rev( X, [A|Y], Z ).

big_plus( A, X, Y ) :- true | bigp( X, Y, A, 0 ).

bigp( [X|Xs], [Y|Ys], A, C ) :- TmpA := X+Y+C |
    bigp1( Xs, Ys, A, TmpA ).
bigp( [], [Y|Ys], A, C ) :- TmpA := Y+C |
    bigp1( [], Ys, A, TmpA ).
```

```
bigp( [X|Xs], [], A, C ) :- TmpA := X+C |
    bigp1( Xs, [], A, TmpA ).
bigp( [], [], A, 0 ) :- true | A=[].
bigp( [], [], A, C ) :- C>0 | A=[C].

bigp1( Xs, Ys, As, TmpA ) :- TmpA >= 100000,
    D := TmpA - 100000 |
    C = 1,
    As = [D|A2],
    bigp( Xs, Ys, A2, C ).
bigp1( Xs, Ys, As, TmpA ) :- TmpA < 100000 |
    C = 0,
    As = [TmpA|A2],
    bigp( Xs, Ys, A2, C ).
```