Experience with the Super Monaco Optimizing Compiler

Evan Tick, Bart C. Massey and Jim S. Larson

CIS-TR-95-07
March 1995

Abstract

"Super Monaco" is a shared-memory multiprocessor implementation of a flat concurrent logic programming language. The system evolved from the earlier Monaco project, and retains, by-and-large, the Monaco intermediate abstract machine. Over the past two years, the compiler and runtime system were modified, incorporating a number of new features improving robustness, flexibility, maintainability, and performance. The optimizing compiler, written in KL1, takes high-level programs and produces intermediate code for the Monaco abstract machine. An "assembler-assembler" converts a host machine description into a KL1 program which translates Monaco intermediate code into target assembly code. There are currently two intermediate code translators: one for SGI MIPS-based hosts, and another for Sequent Symmetry 80386-based multiprocessors.[1] This paper discusses the compiler design and our experience building it. A cost/benefit analysis of the compiler optimizations is given, with a comparison to similar systems.

This article was submitted to the *Journal of Logic Programming*, special issue on High Performance Implementations of Logic Programming Systems.

Department of Computer and Information Science
University of Oregon

Contents

1 Introduction

Monaco is a high-performance, shared-memory multiprocessor implementation of a subset of KL1, a flat concurrent logic programming language [27]. "Super Monaco" is a second-generation implementation of this system, consisting of an evolved intermediate instruction set, a new assembler-generator, and a new runtime system. It incorporates the lessons learned in the first design [33, 34] and improves upon its predecessor with better memory utilization (via a 2-bit tag scheme, the use of 32-bit words and garbage collection), and several other runtime system innovations [21].

Three strategies were key to the Super Monaco design: 1) native code, rather than C code, generation; 2) real parallel execution model, and 3) decision graph, rather than thread, generation. Our motivation was to produce a high-performance shared-memory multiprocessor implementation from scratch, to retain full control over the translation, and to understand all the difficulties first-hand. We wanted the resulting programs to execute in parallel to take advantage of the concurrent semantic model. Finally, we subscribed to the efficiencies afforded by compiling procedures into decision graphs, rather than breaking them up into threads to avoid latencies. This decision hinged on our shared-memory target, and was motivated by earlier work by Crammond [5].

The compiler was designed with the philosophy of generating a low-level abstract instruction set for conversion into native code. We felt at the time that by modeling our abstract machine instructions after RISC instructions, the final assembly would be more direct and introduce less overhead than, for example, a WAM-like instruction set. Other load-store instruction sets for logic programming languages exist (e.g., [12, 13, 14, 19, 26, 28, 37]), although they have been primarily designed for specialized hardware. Some of these (and other systems, based on C code generation) are described in Section 7. In hindsight, we discovered our strategy was too extreme, and we later modified the instruction set as discussed in later sections.

In summary, Super Monaco shows uniprocessor and multiprocessor execution performance competitive with systems implementing similar languages. Although for very small programs Super Monaco performs 2.5 times slower than systems that compile into C, for larger programs the differences average to zero. Super Monaco also functions as a testbed for experimentation both with innovative static analyses (e.g., [22]) and runtime systems. Another 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.

This paper is organized as follows. Section 2 gives an overview of the compiler. Section 3 introduces the assembler-assembler. Section 4 discusses the Monaco abstract machine instruction set. Section 5 details key compilation phases. Section 6 gives a cost/benefit analysis of compiler optimizations based on empirical benchmark evaluation. The literature is reviewed in Section 7. Conclusions and drawn in Section 8.

2 Compiler Overview

The Super Monaco compiler translates programs written in a subset of KL1 [16] to Monaco intermediate code. The compiler has been continually upgraded from its first release [34]. The most significant additions, with respect to performance, have been type inferencing and improved code generation of control flow. The compiler consists of about 2500 lines of "front-end" KL1 code which translates source programs to an intermediate form with explicit decision graphs [18], and about 4500 lines of "back-end" KL1 code which compiles this intermediate form. The process by which a Super Monaco executable is produced is described in Figure 1. A machine description written in a special language is transformed into a template-based translator. The KL1-subset source program is compiled to our intermediate form, which is then translated into native assembly code.

The kernel Super Monaco compiler is summarized in Figure 2. The pipeline follows a traditional

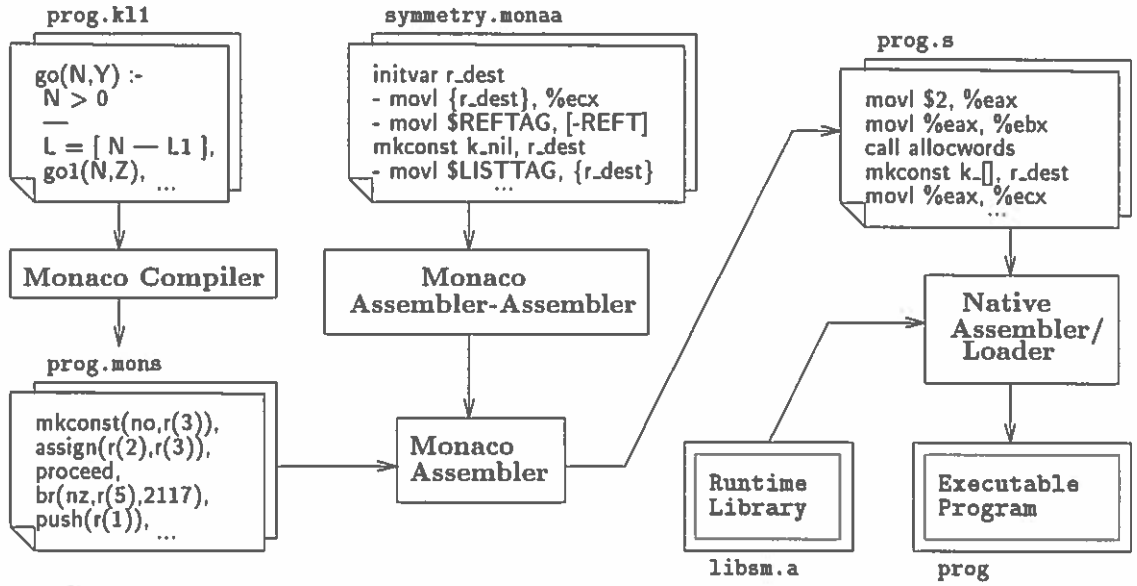

Figure 1: Overview of the Super Monaco System

Figure 2: Monaco Compiler Organization (Main Phases Shown)

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 and flattens the program, does limited type inference, and generates decision graphs using Kliger's algorithm [18]. These graphs and trees are fed to a code-generation phase which produces rudimentary abstract machine code, consuming an arbitrary number of pseudo-registers. Type information is used to avoid type checking in some cases. The code is then passed through an optimizer which builds a flow graph of basic blocks, and performs memory allocation coalescence.

During common subexpression elimination analysis, type and dereferencing information is propagated through the flow graph. At this point, macro-instructions are resolved, and redundant computations are recognized and eliminated. Deadcode elimination is a minor pass not shown 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) then perform jump-to-jump short circuiting, dead block removal, branch removal, code flattening, peephole optimization, and register move chain squashing (in that order). The final output is ready for translation to native code.

The number of registers consumed in the target program is limited by a compiler parameter (so that the registers in the intermediate language can be mapped onto general-purpose machine

registers of the native-code target) but is otherwise machine-independent. This scheme leads to good portability, while also allowing some experimentation, such as artificially restricting register usage to measure performance impacts, or implementing "extra registers" using memory locations.

The intermediate code design was originally targeted toward RISC-based microprocessors, and some vestiges of this decision remain in the compiler. For example, the assumption of a reasonably large number of general-purpose registers (if fewer than about 16 registers are available, code quality degrades substantially) requires the Sequent Symmetry implementation, with only four general-purpose registers available, to implement all of its registers as an array in memory. The original Monaco assumption that condition-codes are not available as the result of arithmetic and logical computations led to implementation inefficiency on non-RISC architectures, because explicit logical temporaries were generated and tested, consuming both extra registers and extra instructions. This has been fixed by redesigning branch instructions. Overall, the quality of the generated code is high (see Figures 4 and 9).

3 The Monaco Assembler-Assembler

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

The **monaa** translator consists of about four hundred lines of **awk** code, together with a small Bourne shell driver and some **m4** macro definitions. The overall structure of the **monaa** language is that of a simple template expander — no native-code peepholing or other optimizations are currently done, although it is possible that this will change in the future (see Au-Yeung [2] for a formal language description). For each **mona** instruction, one or more non-overlapping parameterized templates are given, together with machine code produced in response to the match. Type information is attached to both the formal and actual parameters to guide matching and expansion. In addition to instruction templates, the **monaa** description provides information about register names and calling conventions, as well as some standard templates for procedure prologues and epilogues, debugging information, and the like. The generated native assembly code follows the C calling conventions for linking with the runtime system, and allows for profiling and symbolic debugging of Monaco assembly code with standard UNIX tools. The **monaa** description for the Sequent Symmetry is about 700 lines of **monaa** code, expanding to about 1300 lines of KL1. The machine-independent KL1 code for **mona** comprises about 3400 lines, including symbol-table management and basic housekeeping functionality.

Some of the **monaa** templates used for current targets are given in Figure 3. Note that the templates of the i386 implementation of the Monaco instructions (a) are somewhat larger than those of the MIPS implementation (b). This is due in small degree to the two-address nature of i386 instructions (as opposed to MIPS three-address instructions), but largely to the fact that the i386 Monaco registers are actually implemented using memory locations. The small number of general-purpose registers available on the i386 forced this implementation, and the Monaco registers thus must be copied to and from real registers in each instruction.

The use of **monaa** has proved to have several advantages: 1) The specialized machine description

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

(a) Symmetry (i386) Templates

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

(b) MIPS Templates

Figure 3: Code Templates for `monaa`

```
deref(r(0),r(3))
br(isntlist,r(3),13)
alloc(4,r(7))
initvarref(r(7),1,r(4))
car(r(3),r(6))
initlistref(r(7),2,r(6),r(4),r(5))
assign(r(2),r(5))
cdr(r(3),r(0))
move(r(4),r(2))
execute(append/3)
label(13)
br(isntnil,r(3),16)
unify(r(2),r(1))
proceed
label(16)
br(isbound,r(3),19)
push(r(0))
label(19)
suspend(append/3)
```

Figure 4: Monaco Intermediate Code for `append/3`

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

4 Monaco Abstract Machine

The Monaco instruction set presents an abstract machine at an intermediate level between source program and target program (native machine code) semantics. The abstract machine consists of a number of independent worker processes which execute a sequence of procedures and update a shared memory area. Each worker has a set of abstract general-purpose registers which are used as operands for Monaco instructions and for passing procedure arguments. Control flow within a procedure is sequential with conditional branching to code labels. Figure 4 shows the Monaco code produced by the compiler for `append/3` (more details below).

4.1 Storage Model

The abstract machine memory consists of a single, shared address space. The implementation and management of the space is beyond the scope of this paper (see Larson *et al.* [20]).

The shared memory area is divided into cells, each of which can contain a Monaco data object, also called a *term*. The taxonomy of Monaco terms is illustrated in Figure 5. All objects are represented as 32-bit words of memory aligned on four-byte address boundaries. This alignment restriction allows the low-order two bits of pointers to be used as tag bits, without loss of pointer range. The four tagged types are *immediates, list* pointers, *box* pointers, and *reference* pointers. Immediates are further subdivided into *integers, atoms,* and box *headers.* Integers have the distinction of being tagged with zero bits, allowing some optimizations to be made in arithmetic code generation. On most architectures, the pointer types suffer no inefficiencies from tagging, since negative offset addressing may be used to cancel the added tag. There is only one mutable object type — the *unbound variable*, represented as a null pointer with a reference pointer tag. When a variable is bound, its value is changed to the binding value. The design rationale for the runtime data layout is given in Larson *et al.* [21].

4.2 Instruction Set

Committed-choice languages [27] differ from Prolog in several ways, leading to abstract machine definitions that differ from the WAM for efficiency reasons. First, committed-choice languages 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, a binding protocol is needed. Third, there is a wide gap in memory-usage efficiency between concurrent and sequential languages. Parallel Prolog can exploit stacks because of the inherently sequential nature of their threads. 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, whereas 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, the process management of committed-choice languages, i.e., enqueueing, suspending, and resuming operations, is frequent and expensive.

The instruction set consists of about sixty operations, summarized in Table 1. In the table, R_s, R_{s1}, and R_{s2} denote source registers, R_d denotes a destination register, n denotes an integer constant, and F/A is the name of an executable procedure. The instructions take constants or registers as their arguments and return their results in registers. There is no explicit access to the shared memory except through operations which access the fields of aggregates. The static machine instruction counts given in Table 1 vary with the specific abstract instruction, and do *not* include any runtime system subroutine calls.

The operations are broadly categorized as: 1) Data constructors for each data type (constant, list, struct, goal record, variable). 2) Data manipulators for accessing the fields of aggregates. 3) Arithmetic operations. 4) Predicates for testing the types of most objects and for arithmetic comparisons. Predicates store the truth value of their result in a register. 5) Control instructions. 6) Interfaces to runtime system operations for assignment, unification, suspension, and scheduling. 7) Instructions for manipulating the suspension stack. The instructions take constants or registers

Monaco Instruction	80386 instr†	MIPS instr†	Semantics
data constructors			
alloc($Size, R_d$)	17‡	18‡	allocate heap by *Size* cells
initgoalref(R_b, *Off*, *Size*, F/A, R_d)	5	6	initialize a goal record
initlistref(R_b, *Off*, R_{s1}, R_{s1}, R_d)	7	4	initialize a list
initstructref(R_b, *Off*, *Size*, R_d)	4	4	initialize a vector
initvarref(R_b, *Off*, R_d)	5	6	initialize a variable
mkconst(⟨*const*⟩, R_d)	1	1	$R_d :=$ *const*
mkstruct(*Size*, R_d)	17‡	18‡	$R_d :=$ ptr to vector of *Size* cells
mkgoal(*Size*, *Proc*, R_d)	18‡	20‡	$R_d :=$ ptr to goal record
data manipulators			
move(R_s, R_d)	2	1	$R_d := R_s$
deref(R_s, R_d)	9‖	6‖	$R_d :=$ dereference of R_s
car(R_s, R_d)	3	1	$R_d :=$ head of R_s
cdr(R_s, R_d)	3	1	$R_d :=$ tail of R_s
sref(R_s, n, R_d)	3	1	$R_d :=$ value of n^{th} slot in vector R_s
sset(R_s, n, R_d)	3	1	n^{th} slot in vector $R_d := R_s$
ssize(R_s, R_d)	5	3	$R_d :=$ size of vector R_s
arithmetic and predicates			
iadd(R_{s1}, R_{s2}, R_d)	3–6	1–2	integer arithmetic (isub, idiv, imod, imul)
incr(R_s, R_d)	3	1	integer arithmetic (decr)
iand(R_{s1}, R_{s2}, R_d)	3	1–2	bitwise arithmetic (ior, ixor)
ineg(R_s, R_d)	3	1–2	bitwise arithmetic inot)
ieq(R_{s1}, R_{s2}, R_d)	6	2	comparison (ige, igt, ile, ilt, ineq, eq, neq)
isatom(R_s, R_d)	5–9	2–7	type compare (isbound, isunbound, isint, islist, isnil, isref, isstruct, isimm)
isempty(R_d)	5	4	suspension stack empty?
control			
br(a,*Label*)	1	1	jump to *Label*
br(*cond*,R_s,*Label*)	2	1	branch to *Label* (*cond* = n, p, z, nz)
br(*cond*,R_s,*Label*)	2–8	2–5	branch to *Label* (*cond* = islist, isntlist,...)
br(eq(⟨*const*⟩),R_s,*Label*)	2–10	1–6	if (R_s = *const*) then branch to *Label* (neq)
br(igt(⟨*const*⟩),R_s,*Label*)	5	1	if (R_s = *const*) then branch to *Label* (ige,...)
br(eq,R_{s1}, R_{s2}, L)	5	5	if ($R_{s1} = R_{s2}$) then branch to L (neq)
br(igt,R_{s1}, R_{s2}, L)	8	5	if ($R_{s1} = R_{s2}$) then branch to L (ige,...)
process management			
enqueue(R_s)	3°	4°	push goal on ready queue
proceed	2	2	complete process
execute(F/A)	2	4	execute process F/A
punify(R_{s1}, R_{s2}, R_d)	7°	6°	passive unify
unify(R_{s1}, R_{s2})	7°	6°	active unify
push(R_s)	4	9	push address onto suspension stack
suspend(F/A)	8	11	suspend process F/A

† static template size.
‡ includes instructions for memory allocation subroutine.
° calls a runtime routine.
‖ loops.

Table 1: Super Monaco Abstract Instruction Set

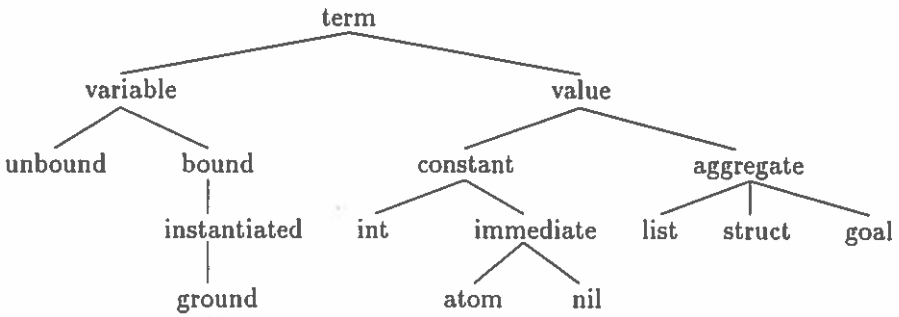

Figure 5: Monaco Object Taxonomy

as their arguments and return their results in registers.

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 as calls to the runtime system. 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 runtime system call frequency.

Each data constructor serves to batch up allocation requests into a large block, and then initialize smaller sections of the block. Batching up the frequent allocation requests increased performance on standard benchmarks (see Section 6). In addition, aggregates which are fully ground at compile time are statically allocated in the text segment of the assembled code. This decreases execution and compilation times. Note that variables reside outside of structures, which will facilitate future optimizations concerning local memory reuse.

Unification is included in the last instruction category. Passive unification verifies the equality of *ground* values. An attempt to passively unify a term containing uninstantiated variables will result in suspension of the process until those variables become instantiated.[2] Active unification, on the other hand, will bind variables to other variables or to values in order to ensure equality of terms. As is customary in logic programming implementations, no "occurs check" is performed during unification for efficiency reasons. Variables are bound through assignment operations or active unification.

The instruction set was modeled after a reduced instruction set (RISC) architecture, on the theory that such small instructions may be easily and efficiently translated to native RISC instructions with a simple assembler. This is the case for the MIPS port, where many Monaco instructions translate to single MIPS instructions, as shown in Table 1. However, the Monaco instruction set has been evolving toward more complex instructions, as frequent idioms are identified and coalesced. There are several reasons for this trend: 1) Intermediate instructions at too low a level violate abstraction barriers between the intermediate code and the machine-level data layout and runtime system data structures. 2) As the amount of work per instruction gets larger, more machine-specific optimizations can be made in the `monaa` code templates. 3) There is no reason to equalize the

[2] This is in contrast to systems such as JAM Parlog [6], which also verify the equality of terms in which uninstantiated variables are bound together. For example the program `equaltest(X,X).` will succeed with two unbound arguments in Jam, whereas in Monaco, such a query will suspend, waiting for the arguments to be instantiated. The differences in these semantics have implications on the efficient implementation of assignment and mode analysis, hence our choice.

amount of work done per instruction or to standardize instruction formats, as there is with RISC architectures. 4) If the native target is not a good match for the Monaco instruction set, a simple template-expanding assembler will produce much better native code for a more complex instruction than for a sequence of simple instructions. (This is in contrast to systems such as [13], a sophisticated multi-level translation scheme which produces good code by intelligent generation of very simple intermediate instructions.)

5 Compiler Internals

There are 11 phases in the compiler: 1) source input; 2) type analysis; 3) decision graph generation; 4) code generation; 5) basic block generation; 6) common subexpression elimination (CSE); 7) live range analysis; 8) dead code elimination; 9) register allocation; 10) branch shorting, peepholing and register chain shorting, and 11) assembly output. Type analysis is the only global analysis; everything else is local to a procedure. In other words, basic blocks are produced on a per procedure basis, and dataflow analyzed for CSE, live ranges, and register allocation.

In the following subsections, we give more details about the compilation process and the algorithms used. Rather than giving formal specifications of the algorithms, our discussion is informal, and discusses advantages and disadvantages of our approach. We follow this by empirical cost/benefit analysis of the key optimizations in Section 6.

5.1 Type Inference by Abstract Interpretation

Type inference is performed by a "poor man's" abstract interpreter. Types are derived for head arguments only. For these, the domain of interest is: `unbound`, `bound`, `bound-to-integer`, `bound-to-atom`, `bound-to-list`, `bound-to-nil`. `unbound` means that nothing is known about the argument's binding. `bound` means that the argument is guaranteed to be bound upon procedure invocation, but that the type of the binding is unknown. Although the domain is simplistic (for example, type propagation through the subterms of complex terms is not modeled) interpretation is fast, and some valuable information is derived (see Section 6).

First the source program is converted into an abstract call graph implemented as a tableau. Each tableau entry represents a procedure, holding the head arguments, their current (abstract) substitutions, body calls and their current call substitutions, and a single queued abstract invocation. Abstract substitutions for head arguments are kept as *vectors* of domain values throughout the interpretation. A vector contains one substitution for each parent (caller). A vector is reduced into a scalar substitution, via abstract unification, when setting up arguments for body calls. Abstract unification is defined in the obvious way, for instance the vector [`bound-to-atom`,`bound-to-integer`] reduces to `bound`.

Body call substitutions can be initialized to fixed domain values in certain cases. For example, if a guard tests `X>3`, then for a body goal `f(X)` we know that `X` is `bound-to-integer`. This is the key to type propagation.

The interpretation proceeds by enqueuing body calls in the *callee's* tableau entry. If a previous call is already enqueued there, the two calls are merged. If an abstract reduction does not change the head argument substitutions, then the tableau entry is marked "fixed." A global fixed-point is reached when all entries with enqueued calls are fixed.

Our experience with this abstract interpreter has been positive. It took about three days to implement, and consists of 1039 lines of KL1, only 27% of which is the interpreter (the rest is for creating the initial tableau and annotating the source program with the resulting types). The type information is used during code generation to strength-reduce the decision graphs, which are described next. The design is extensible and future work will be focused on increasing its accuracy,

e.g., by including the domain element list-of-integer and the corresponding rules for creating initial substitutions.

5.2 Decision Graph Generation

Decision graphs have been shown by Kliger [18] to be an 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 formal algorithm is given in Tick and Korsloot [35]. For each procedure, a canonical normalized form is produced. The graph is then generated; each node is a test (e.g., $X > Y - 3$), and edges are valuations of a test (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 the graph into a sequence of triples suitable for optimization.

The key element of the decision graph generation algorithm is the computation of *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 the proofs by table lookup.[3] For example, for integers k_1 and k_2 such that $k_1 > k_2$, the constraint $X > k_1$ implies the constraint $X > k_2$. A set of these relations has been found to be quite effective in allowing optimized graph generation. Complex inferences (such as transitivity) cannot be made. However, we have not seen such complexity in typical programs.

A core function within the decision graph construction algorithm is *indexing* which chooses the next test to generate (from the root downwards) from among a set of candidates. The indexer schedules a test higher in the graph when more clauses "care" about the test, a purely syntactic metric.[4] The heuristics of caring are quite complex [35]. The decision graph generator consists of 1253 lines of KL1 code, 40% of which defines the indexing heuristics. For experimentation purposes, a naive decision graph generator was also implemented within Monaco, which schedules *every* guard seen in the program, without sharing tests. The implementation was very simple (149 lines of KL1 code), but performance is poor. These cost/performance tradeoffs are evaluated in Section 6.

Consider the `check/8` procedure in the queens benchmark, listed in Figure 6. This example illustrates some of 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)`, `integer(C)`, and `integer(D)` are included in clauses 2–4. The graph produced is shown in Figure 7.

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 (integer checks in this case) that are known *apriori* from flow analysis [7]. Since we have not implemented call forwarding in Super Monaco, this concern is not relevant and 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 2 (node 4) requires *three* arithmetic inequalities. Ideally, node 3 should commit to clause 2. 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, $P - C = D$

[3] The Aquarius Prolog compiler [37] simplifies formulae in a similar manner. Both Monaco and Aquarius use about 50 rules of comparable complexity [35].

[4] Debray *et al.* present a decision tree generation scheme exploiting dynamic caring information.

```
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 6: Normalized Procedure **check/8** from queens

```
switch(X,
  [case([],commit(clause 1))
   case('.'/2,
      ask(integer(P),
          yes(ask(integer(C),                                    (1)
                  yes(ask(integer(D),                            (2)
                          yes(ask((C-P =\= D),
                                  yes(ask((P-C =\= D),
                                          yes(commit(clause 4))
                                          no(go(7))              (3)
                                      7:other(go(6)))),
                                  no(commit(clause 3)),
                              6:other(
                                  ask((P-C =:= D),
                                      yes(commit(clause 2)),     (4)
                                      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 7: Stylized Decision Graph for **check/8**

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.

The decision graphs are generated without regard to type information, which is embedded in the graph via variable annotations. During code generation the type information is used to avoid test generation. In the full **queens** benchmarks evaluated in Section 6, **check/8** arguments **C** and **D** are inferred to be **bound-to-integer**, allowing nodes 1 and 2 to be removed. 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.

5.3 Code Generation

Code generation is driven by the decision graph grammar in a mechanical fashion. The strategy employed is to keep code generation as simple and direct as possible, at the later expense of cleaning up inefficient code sequences with dataflow analysis and other backend optimizations (Sections 5.4 and 5.5). The resulting code is somewhat 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. 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 generated 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 [7], are 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.

To illustrate our code generation techniques, Figure 8 shows a single clause procedure, its decision graph (right side) annotating the initial code generated (left side). There are several interesting points. Naive translation of the continuation linkage in the decision graph creates many branches: there are 18 control instructions out of 29 total instructions! Of these, 2 branches are dead and 5 other unconditional branches are the target of a previous branch. One branch (to label 11) was artificially placed in the body to split the code into smaller blocks, facilitating register allocation (see Section 5.4). In all these cases, later branch squashing cleans up the flow (Section 5.5).

Note the code generated for ask tests. As was mentioned in the previous section, the code generated must test if the operand is unbound and if so, push the operand on the suspension stack and jump to the "other" continuation. Type information circumvents such code, e.g., if **$PARAM(3)** was annotated **bound-to-integer**, then instructions 2–5 would not have been generated. For more complex procedures, types and branch-chain squashing are not sufficient to get quality code. For such cases, we need to derive common subexpressions using dataflow analysis, as is described next.

5.4 Common Subexpression and Dead Code Elimination

Dataflow 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 [1] is used; both the usual branches and 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). The flow graph 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

```
f( X, Y, Z ) :- Z > 3 | f( Z, Y, X ).
```

```
deref($PARAM(3),$REG(4)),
br(isint,$REG(4),4),
br(isbound,$REG(4),5),
push($REG(4)),
br(a,2),
 4:deref($PARAM(3),$REG(5)),
   br(igt(3),$REG(5),7),
   br(isbound,$REG(5),10),
   push($REG(5)),
   br(a,3),
10:br(a,8),
    7:commit,
      move($PARAM(3),$REG(6)),
      move($PARAM(2),$REG(7)),
      move($PARAM(1),$REG(8)),
      br(a,11),
   11:move($REG(6),$OUTPARAM(1)),
      move($REG(7),$OUTPARAM(2)),
      move($REG(8),$OUTPARAM(3)),
      execute(f/3),
      br(a,9),
    8:br(a,3),
      br(a,9),
    3:br(a,2),
    9:br(a,1),
    5:br(a,2),
      br(a,1),
    2:br(a,1),
    1:suspend(f/3)
```

```
ask(
    integer( V3 ),
    |
    |
    |
    yes(
    |   ask(V3 > 3,
    |       |
    |       |
    |       |
    |       |
    |       yes(
    |       |
    |       |
    |       |
    |       |                V3,
    |       |                   V2,
    |       |                      V1
    |       |            f(        ) ) )
    |       |
    |       no( go(3) ),
    |       |
    |       other( 3:go(2) ) ) ),
    |
    no( go(2) ),
    |
    other( 2:go(1) ) ),
  1:suspend( f/3 )
```

Figure 8: Simple Code Generation Example

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)`. We may then derive the new information that `R4 <- deref(car(deref(R1)))`.

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 `br(isint,R6,L)`, will rewrite the latter to be `br(isint,R5,L)`. Dead code elimination will then cancel the `move`. 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. For example, `br(nz,R8,L)`, 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. The branch will thus 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 register liveness information for 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. Given the previous analysis, this phase is trivial. Next, we allocate registers for each procedure.[5] The local allocation method used is based on the liveness of the registers, and is performed on a basic-block granularity to match live-range analysis. The most live name (i.e., the name live across the largest number of basic blocks) 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 quality 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 basic blocks (in the limit, into individual 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. For our MIPS port, 16 real registers are allocated (the other 16 are reserved by MIPS convention or by the runtime system).

Our allocator averaged 7.3 registers for 37 benchmark procedures considered in Tick and Banerjee [34]. Trivial procedures typically required 2–7 registers, whereas more complex procedures typically required 11–17 registers. Future work includes implementing an interprocedural register allocator based on cooperation across procedure call boundaries.

Figure 9 shows the results of compiling the program `f(a(b(c(_))))`. The final code is the product of all of the backend optimizations. Notably, CSE rewrites instructions 1–4 to moves, which are later collapsed. The `structref` instructions are instantiated to `sref` instructions by the dataflow analysis (as opposed to `car` and `cdr` instructions had they been list references). Instructions 5–8 are removed by jump chain collapsing. Instructions (*) are removed during a topological sort of the flow graph.

5.5 Miscellaneous Optimizations

Final phases include shorting of 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. Finally, the register move chains are shorted. Currently the peepholer is limited: its primary function is to attempt CSE of spill sequences. This is inherent to the well-known problem of where to allocate registers: before or after CSE. Since we allocate after, 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 [30], to better solve the problem.

Worthy of comment is the register move chain shorting algorithm. As mentioned in Section 5.3, an invariant obeyed during code generation is that a pseudo-register cannot be redefined after it is used. This rule simplifies dataflow analysis for CSE. Unfortunately, the Monaco abstract machine's calling convention is to pass arguments during a call through fixed registers. Thus the incoming and outgoing arguments must share the same *real* registers, but not the same *pseudo* registers. We introduce special pseudo-registers `$PARAM` and `$OUTPARAM` to solve this problem. An example of their use is shown in the tail call in Figure 8. This technique allows the early code to obey the invariant, and is cleaned up during register allocation, by considering $\texttt{\$PARAM}(k) = \texttt{\$OUTPARAM}(k)$.

The trick has a drawback: it means that the basic block containing a tail recursive call will usually have very poor register assignment. To fix this, after final flattening of the basic blocks,

[5] One of our current areas of research involves interprocedural register allocation utilizing sequentialization of threads [22].

```
    deref(r(1),r(2))
    br(eq(a/1),r(2),5)
    br(a,2)                 (*)
5:  deref(r(1),r(4))        (1)
    structref(r(4),1,r(5))
    deref(r(5),r(3))
    br(eq(b/1)),r(3),6)
    br(a,3),                (*)
6:  deref(r(1),r(7))        (2)
    structref(r(7),1,r(8))  (3)
    deref(r(8),r(9))        (4)
    structref(r(9),1,r(10))
    deref(r(10),r(6))
    br(eq(c/1)),r(6),7)
    br(a,4)                 (*)
7:  proceed
4:  br(isbound,r(6),8)
    push(r(6))
    br(a,8)
8:  br(a,3)                 (5)
3:  br(isbound,r(3),9)
    push(r(3))
    br(a,9)
9:  br(a,2)                 (6)
2:  br(isbound,r(2),10)
    push(r(2))
    br(a,10)                (7)
10: br(a,1)                 (8)
1:  suspend(f/1)
```

(a) before optimizations

```
    deref(r(0),r(1))
    br(eq(a/1),r(1),3)
1:  br(isbound,r(1),2)
    push(r(1))
2:  suspend(f/1)
3:  sref(r(1),2,r(4))
    deref(r(4),r(2))
    br(eq(b/1),r(2),5)
4:  br(isbound,r(2),1)
    push(r(2))
    br(a,1)
5:  sref(r(2),2,r(5))
    deref(r(5),r(3))
    br(eq(c/1),r(3),6)
    br(isbound,r(3),4)
    push(r(3))
    br(a,4)
6:  proceed
```

(b) after optimizations

Figure 9: Example of CSE and Backend Optimizations

each procedure is reversed and scanned, shorting its moves. We call the specific algorithm "tail squashing" because it is most effective in tail call blocks.

6 Cost/Benefit Analysis

Super Monaco was evaluated for two suites of benchmarks executing on a Sequent Symmetry S81 with 16MHz Intel 80386 microprocessors. The first set, consisting of six small programs (e.g., [34]), was used primarily for comparisons with results in the literature. The second set, containing six larger programs, allowed a more realistic cost/benefit analysis of the compiler. Benchmark **cubes** finds solutions to a combinatorial puzzle problem; **semigroup** computes a Brandt semigroup [32]; **waltz** implements Waltz's line-drawing constraint satisfaction algorithm [32]; **bestpath** is a concurrent algorithm for finding the shortest path in a graph [32]. **wave**, written by I. Foster, computes an iterative sum around a multidimensional torus; **life**, written by A. Goto, plays the game of life; **absearch** is an alpha-beta-pruned minimax game tree search.

Table 2 compares the uniprocessor performance of SICStus Prolog (v2.1),[6] Jam Parlog (v1.5.9), Strand (Buckingham release),[7] `jc`,[8] original Monaco, KLIC (uniprocessor v1.500), and Super Monaco

[6] Native-code SICStus is not available for Symmetry.

[7] This release is rather old, we are attempting to procure a newer version.

[8] Measurements taken from Gudeman *et al.* [10], an earlier implementation of Janus. The latest version v113+r102 could not produce optimized code for the Symmetry, because of a documented GCC bug [11]. Without optimization, performance comparison is meaningless in this context. When we find a way around this bug we intend to update the

benchmark	SICStus	Strand	(old) Monaco	jc	Jam	KLIC	Super Monaco	SM:KLIC
hanoi(14)	4.6	5.8	4.4	1.2	5.6	0.6	2.3	3.83
nrev(1000)	21.0	34.3	19.2	3.9	38.2	5.9	11.9	2.02
pascal(200)	14.4	21.8	9.0		19.0	1.7	4.1	2.41
primes(5000)	29.2	38.7	12.8	6.2	39.5	4.4	9.3	2.11
queens(10)	106.0	25.4	43.4	39.4	140.6	10.4	28.3	2.72
cubes(6)	113.9				151.4	15.5	38.0	2.45
semigroup					125.6	85.9	140.2	1.63
waltz					87.7	18.8	27.2	1.45
bestpath						193.8	80.2	0.41
wave(8,8)						11.6	7.4	0.64
life(20)					51.2	29.6	20.2	0.68
absearch						8.4	19.5	2.31

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

(v1.0). All times are the best of several runs, using the sum of user- and system-level CPU times. Because garbage collection (GC) is hidden within these systems, we cannot normalize the measurements for GC effects. We believe it unlikely that GC could be more than 25% of measured time for these benchmarks.

SICStus, given solely as a baseline, is measured only for the small benchmarks that are directly translatable into Prolog. Some benchmarks were not measured on some systems, either because of inaccessibility or system bugs. Super Monaco was found to outperform Strand and Jam in a uniprocessor configuration by factors ranging from 1.6 to 4.0 (except for queens, for which Strand performed remarkably), and to maintain such ratios for moderate numbers (1–16) of processors [34]. In all cases, Super Monaco improves on the performance of the previous Monaco system by factors of about two.

Because the jc measurements are incomplete and old, we focus our comparison on KLIC.[9] Compared to KLIC, Super Monaco performance is especially good for the larger, more realistic benchmarks, where the geometric mean slowdown is 1.0, i.e., dead even. Still, we show a geometric mean slowdown of 2.5 for the small programs. How much of this is due to compiler-generated inefficiencies, as opposed to runtime support, is unclear at this time. With over 50% of program execution time in the runtime system, the compiler cannot be held responsible for more than the remainder. Within this slice, we think that register allocation is the weakest link, especially on the 80386. Future performance evaluation on the SGI MIPS-based multiprocessor is needed.

Table 3 gives the multiprocessor execution times of Super Monaco. Times are for the longest running processor from the beginning of the computation until termination. Some of these benchmarks were executed for larger data than in Table 2. The superlinear behavior of bestpath is because the algorithm is nondeterministic. absearch is inherently sequential and thus achieves only slowdown on increasing numbers of processors. Disregarding the outliers, the geometric mean speedup on 16 processors for the small and large benchmarks was 12.7 and 10.6 respectively. These speedups are comparable with Jam and the original Monaco, while absolute performance is superior (see Larson *et al.* [21] for detailed measurements).

Table 4 gives the execution and compilation times for each of the five optimization levels of the

measurements presented here.

[9] See Chikayama *et al.* [4] for a an in-depth comparison between KLIC and jc.

benchmark	Processors					
	1	2	4	8	12	16
hanoi(17)	19.38	10.15/1.9	5.16/3.8	2.62/7.4	1.78/10.9	1.36/14.3
nrev(1200)	17.23	9.90/1.7	5.62/3.1	3.37/5.1	2.42/ 7.1	1.97/ 8.7
pascal(400)	27.26	14.04/1.9	7.34/3.7	3.84/7.1	2.68/10.2	2.14/12.7
primes(9000)	24.20	13.20/1.8	7.10/3.4	4.11/5.9	3.01/ 8.0	2.43/10.0
queens(10)	26.79	13.86/1.9	7.02/3.8	3.57/7.5	2.41/11.1	1.85/14.5
cubes(6)	36.75	18.69/2.0	9.45/3.9	4.75/7.7	3.20/11.5	2.43/15.1
semigroup	140.14	71.62/2.0	38.81/3.6	21.24/6.6	16.26/ 8.6	12.25/11.4
waltz	27.24	14.18/1.9	7.26/3.8	3.82/7.1	2.69/10.1	2.28/11.9
bestpath	80.17	35.02/2.3	18.43/4.4	10.96/7.3	6.25/12.8	5.84/13.7
wave(12,12)	46.25	26.76/1.7	13.80/3.4	7.52/6.2	5.37/ 8.6	4.57/10.1
life(20)	20.21	11.11/1.8	5.88/3.4	3.23/6.3	2.46/ 8.2	2.18/ 9.3
absearch	19.47	26.56/0.73	28.85/0.67	29.63/0.66	29.79/0.65	30.22/0.64

Table 3: Super Monaco Multiprocessor Performance (Seconds/Speedup, Symmetry)

compiler. Each optimization level builds upon optimizations in previous levels. Each benchmark program was compiled and run five times and the minimum measurement was chosen. Programs were compiled on a Sun MP4 (bootstrapping the compiler using PDSS [15]) and executed on a Symmetry. For a given benchmark, for a given optimization level i (column), the percentage $(T_i - T_{i-1})/T_{i-1}$ is given, representing the cost (if positive) or benefit (if negative) of the optimization over the previous optimization.

The first point to note is that performance results for small programs do not reflect those of larger, more complex programs. Usually the smaller benchmarks see much larger gains. Limiting our comments to the larger programs, type inferencing was a disappointment: performance was flat, though cost was negligible. Dataflow analysis cost 8% compilation and gained 7% execution. Decision graph were a win-win proposition, having negative cost and postive gain.[10] For example, the -27% average compilation cost represents a *gain* in compilation efficiency, because the indexing produces simpler decision graphs that in turn decrease backend processing. In contrast, saving frontend compilation time by forgoing indexing *costs* 1.2% over naive graph generation.

Table 5 shows the compilation time broken down by function, for the highest optimization level. Computing live ranges is most costly, followed by register allocation and input/output. Other functions are relatively inexpensive. The main reason the global flow analysis algorithms are inefficient is because we do not simplify the flow graph early enough. It would pay to invest in early simplification (before CSE). One must be careful, however, because certain basic block splits were purposely made to improve register allocation. Another weakness in the compiler is seen in bestpath: there is an 87-clause procedure indexed on one (integer) argument. Unfortunately, a general decision graph is constructed for this procedure, requiring 25% of compilation time, and generating large amounts of code which consume additional compilation time during later phases. A single switch instruction is finally created for this procedure in the peepholer, but far too late to save compilation time.

A final note about comparative systems' compilation time: although no extensive measurements were collected, we observed that Super Monaco compilation time lies between Jam and `jc`, and closer to Jam. The Jam non-optimizing compiler (to byte code) is rather fast, whereas the `jc` compiler is

[10] This is not an entirely fair statement. For example the JAM Parlog compiler is significantly faster than Super Monaco, at any optimization level. The main reason for this is JAM's lack of flow analysis and register allocation. Decision graphs have negative cost in Super Monaco because they remove large chunks of code which then are not processed by later optimizations — optimizations which JAM would not perform in any case.

benchmark	no opt.	dec. graph w/o index	dec. graph w/ index	dataflow analysis	type inference
	execution time (seconds/%, Symmetry)				
hanoi	19.99	20.02/ 1.1%	19.99/ -1.0%	19.43/ -2.8%	19.38/-0.3%
nrev	25.93	25.95/ 0.0%	21.97/-15.3%	17.24/-21.5%	17.23/-0.0%
pascal	46.82	43.63/ -6.8%	43.15/ -1.1%	28.54/-33.9%	27.26/-4.4%
primes	39.92	35.92/-10.0%	36.89/ 2.7%	25.19/-31.7%	24.20/-4.2%
queens	46.95	37.61/-19.9%	35.81/ -4.8%	27.77/-22.5%	26.79/-3.5%
cubes	66.27	50.44/-23.9%	48.28/ -4.3%	37.10/-23.2%	36.75/-0.9%
mean		-9.9%	-4.0%	-22.6%	-2.2%
semigroup	144.79	144.88/ 0.0%	143.31/ -1.1%	140.10/ -2.2%	140.14/ 0.0%
waltz	31.24	31.87/ 2.0%	29.83/ -6.4%	27.00/ -9.5%	27.24/ 0.9%
bestpath	112.34	110.98/ -1.2%	84.47/-23.9%	80.04/ -5.2%	80.17/ 0.2%
wave	57.07	54.55/ -4.4%	49.42/ -9.4%	46.89/ -5.1%	46.25/-1.4%
life	22.23	21.88/ -1.6%	21.56/ -1.5%	20.21/ -6.3%	20.21/ 0.0%
absearch	24.87	26.39/ -6.1%	23.35/-11.5%	20.47/-12.3%	19.47/-4.9%
mean		-1.9%	-9.0%	-6.8%	-0.9%
	compilation time (seconds/%, Sun MP4)				
hanoi	2.95	3.04/ 3.1%	2.75/ -9.5%	3.06/11.3%	3.19/ 4.2%
nrev	2.21	2.15/ -2.7%	1.89/-12.1%	2.02/ 6.9%	1.90/-5.9%
pascal	12.56	11.05/-12.0%	9.49/-14.1%	9.89/ 4.2%	8.91/-9.9%
primes	4.00	3.83/ -4.2%	3.40/-11.2%	3.60/ 5.9%	3.64/ 1.1%
queens	8.42	6.11/-27.4%	5.54/ -9.3%	5.69/ 2.7%	5.43/-4.6%
cubes	15.46	13.55/-12.4%	11.59/-14.5%	12.47/ 7.6%	13.00/ 4.3%
mean		-9.3%	-11.8%	6.4%	-1.8%
semigroup	29.68	23.79/-19.8%	16.09/-32.4%	17.77/10.4%	18.30/ 3.0%
waltz	26.99	29.33/ 8.7%	25.23/-14.0%	26.68/ 5.7%	27.85/ 4.4%
bestpath	105.02	129.58/ 23.4%	63.39/-51.1%	67.70/ 6.8%	68.26/ 0.8%
wave	186.59	166.20/-10.9%	100.28/-39.7%	109.32/ 9.0%	103.37/-5.4%
life	22.97	23.86/ 3.9%	20.68/-13.3%	22.44/ 8.5%	22.96/ 2.3%
absearch	73.55	75.09/ 2.1%	66.70/-11.2%	70.37/ 5.5%	68.10/-3.2%
mean		1.2%	-27.0%	7.7%	0.3%

Table 4: Cost/Benefit of Compiler Optimizations

function	semi	waltz	life	best	wave	search	rest	mean
program input	3.0	3.9	2.4	2.4	1.0	1.9	3.0	2.5
type analysis	3.2	4.8	4.2	1.7	2.2	3.5	5.2	3.5
decision graph	8.7	5.2	7.1	25.6	8.0	7.1	7.5	9.9
code generation	2.8	2.9	3.0	1.3	1.4	2.6	3.3	2.5
basic blocks	6.8	5.4	5.8	5.3	4.4	6.8	6.6	5.9
CSE	6.4	6.0	6.5	6.3	5.1	7.4	7.0	6.4
live ranges	31.4	30.1	34.1	30.0	35.2	32.3	30.9	32.0
dead code	2.6	3.2	4.3	1.2	5.7	4.5	2.2	3.4
register allocation	20.7	18.8	20.7	14.3	29.4	21.5	17.6	20.4
shorting, peepholing	3.9	4.8	4.6	0.6	3.4	3.7	3.9	3.6
program output	10.6	14.9	7.2	11.4	4.3	8.6	12.8	10.0

Table 5: Percentage Compilation Time by Function (Full Optimizations, Sun MP4)

hamstrung by its internal C compilation.

7 Literature Review

Emulation-based real-parallel shared-memory implementations of committed-choice languages include Panda and JAM. Panda [25] was an experimental system at ICOT implementing a subset of FGHC. and utilizing a WAM-like abstract machine instruction set [16]. Jim's Abstract Machine (JAM) [6] is a Parlog emulator including support for Or-Parallel execution of *deep* guards. These systems neither use optimizing compilers nor produce native code.

Strand [9] 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 [36]. 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.

Further restricted from Strand is the original Janus language in which the programmer must declare a single producer and consumer for a stream. Janus has since evolved to be almost identical to Strand [11]. A Janus-to-C compiler `jc` has been developed for uniprocessors, which generates C code [10]. `jc` has several advantages over the Monaco compiler: 1) the backend C compiler can do much better register allocation on the host measured in this paper (an 80386-based Symmetry); 2) the uniprocessor implementation of Janus allows optimizations such as suspension analysis [8] that cannot be easily performed for the real-parallel implementation of Monaco; 3) `jc` has other optimizations not found in Monaco, such as call forwarding [7].

Like jc, Kliger's FCP compiler [17] is targeted to uniprocessors, although it is emulated. 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.

KLIC [4, 24, 23] is a portable, multiprocessor implementation of KL1. Like `jc`, KLIC compiles into C. Because these systems do not produce an intermediate representation, we cannot contrast

them with the Super Monaco instruction set. KLIC, like `jc`, leverages high performance from the backend C compiler. Multiprocessor KLIC (built on PVM) uses explicit task allocation and so it was not appropriate to compare speedups with Super Monaco.

Recent work in logic program compilation for high performance that deserves mention is Mercury [29] a strongly-typed logic programming language and implementation. The language is similar to Prolog and by exploiting types it achieves execution speeds superior to Aquarius [37] and SICStus [3] Prolog. In Super Monaco we retained an untyped language, making comparison difficult. Furthermore, because of our concurrent semantics, mode information would be less directly useful than it is in Mercury, unless suspension analysis is performed.

Finally, we mention RISC-based microprocessor architectures for committed-choice languages: Carmel (e.g., [12]), PIM/i [26], PIM/p [19], and UNIRED-II [28]. These implementations are akin to Monaco, however, they are experiments in specialized hardware, not compilation technology.

8 Conclusions

We have presented the Super Monaco compiler, a shared-memory implementation of flat committed-choice languages. In the spirit of Van Roy [37] and Taylor [31], 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) its backend generates native code, and 4) 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.

The Super Monaco compiler measured in this paper has a few handicaps compared to other implementations. Critically, register allocation, which takes 20% of compilation time, is pointless on Symmetry, where we emulate the register set anyway. Systems exploiting C compilation get far better register performance here. Furthermore, over 50% of program execution time is spent in our runtime system [21], which could benefit from further tuning. Thus compiler optimizations can at best carve out slices from half the pie.

Nevertheless, Super Monaco shows parallel execution performance competitive with similar parallel systems. It can exhibit slower uniprocessor execution than uniprocessor-based systems that compile into C, but this loss of performance can be more than made up by exploiting parallelism. We noticed that larger programs performed similarly on KLIC and Super Monaco, indicating to us that sophisticated C code generation may lose its utility as procedures and programs grow in size. We plan to test this hypothesis for `jc`. If it is true there as well, we believe that the the best approach is to keep the Super Monaco compiler simple (e.g., forgo call forwarding), but improve core functionality, such as register allocation. We expect that on RISC machines the speed differential between native-code and C-code compilers will decrease because of a more level playing field, i.e., sufficient number of general-purpose registers.

Acknowledgements

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

References

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

[2] C. Au-Yeung. A RISC Backend for the 2^{nd} Generation Shared-Memory Multiprocessor Monaco System. Master's thesis, University of Oregon, December 1994.

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

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

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

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

[7] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan. Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 409–420. ACM Press, January 1994.

[8] S. K. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free Logic Programs. In *International Symposium on Logic Programming.* Ithaca, MIT Press, November 1994. Extended version.

[9] 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.

[10] D. Gudeman, K. De Bosschere, and S. K. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*, pages 399–413. Washington D.C., MIT Press, November 1992.

[11] D. Gudeman and S. Miller. *User Manual for jc — A Janus Compiler (Version s113+r102).* University of Arizona, January 1995.

[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] R. C. Haygood. Native Code Compilation in SICStus Prolog. In *International Conference on Logic Programming*, pages 190–204, Genoa, June 1994. MIT Press.

[14] 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.

[15] ICOT. *PDSS Manual (Version 2.52e).* 21F Mita Kokusai Bldg, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, February 1989.

[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 Press, August 1987.

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

[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] 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.

[20] J. Larson, B. Massey, and E. Tick. Garbage Collection in Super Monaco, February 1995. In Progress.

[21] J. Larson, B. Massey, and E. Tick. Super Monaco: Its Portable and Efficient Parallel Runtime System. In *EURO-PAR*, Stockholm, August 1995. Submitted. Also available as University of Oregon Technical Report CIS-TR-95-03.

[22] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In *International Conference on Logic Programming and Automated Reasoning*, number 698 in Lecture Notes in Artificial Intelligence, pages 205–216, St. Petersburg, July 1993. Springer-Verlag.

[23] M. Morita, N. Ichiyoshi, and T. Chikayama. A Shared-Memory Parallel Execution Scheme of KLIC. In T. Chikayama and E. Tick, editors, *Proceedings of the ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, pages 125–134, Eugene, March 1994. University of Oregon Technical Report CIS-TR-94-04.

[24] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed Memory Implementation of KLIC. *New Generation Computing*, 14(3), August 1995. In Press.

[25] 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.

[26] 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.

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

[28] 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.

[29] Z. Somogyi, F. J. Henderson, and T. C. Conway. The Implementation of Mercury, an Efficient Purely Declarative Logic Programming Language. In *Proceedings of the ILPS'94 Post-Conference Workshop on Implementation Techniques for Logic Programming Languages*, pages 31–58. Ithaca, November 1994.

[30] R. M. Stallman. *Using and Porting GNU CC (version 2.6).* Boston MA., September 1994.

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

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

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

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

[35] E. Tick and M. Korsloot. Determinacy Testing for Nondeterminate Logic Programming Languages. *ACM Transactions on Programming Languages and Systems*, 16(1):3–34, January 1994.

[36] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 13(1):3–43, 1994.

[37] 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.