ICOT Technical Report: TR-421

TR-421

Performance of Parallel Logic Programming Architectures

by
E. Tick

September, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Performance of Parallel Logic Programming Architectures

E. Tick*

Stanford University

Institute for New Generation Computer Technology

September 25, 1988

Abstract

This report details the research conducted by the author at the Institute of New Generation Computer Technology (ICOT) for the one-year period beginning September 1987. A comparison between committed and non-committed logic programming language architectures was conducted. KL1, a byte-code architecture for Flat Guarded Horn Clauses (FGHC) and Aurora, a byte-code architecture for Or-parallel Prolog were respectively chosen for this comparison. Three types of parallel emulators were used to measure each of these architectures on a Sequent Symmetry host multiprocessor. The measurements were made on a set of benchmarks developed by the author. Timing emulators were used to measure raw speed of the benchmarks to determine relative performances and speedups. Instrumented (high-level) emulators were used to measure gross characteristics of the benchmarks, such as number of procedures calls and number of instructions executed. Instrumented (low-level) emulators were used to measure detailed characteristics of memory referencing and coherent cache performance, such as miss and traffic ratios. The results of this study indicate that many problems are well-suited to Prolog's powerful unification and backtracking mechanism: however, Aurora is limited at the algorithm level by the primary weakness of OR-parallel search: that processes cannot communicate. On the other hand, most problems can exploit dependent AND-parallelism more easily than OR-parallelism, but the inefficiencies of the KL1 model (no backtracking, excessive use of memory) overshadows the benefits of parallelism. There are a class of problems that perform equally well on both architectures and classes of problems that favor one or the other of the architectures. These results indicate that a high performance system should have backtracking and full unification as well as dependent AND-parallelism.

*Supported by NSF Grant No. IRI-8704576.

Contents

List of Figures

List of Tables

1 Introduction

With the commercial success within the past four years of shared-memory multiprocessors (to the extent of making the front page of the NY Times[42]), the time has finally come to implement high-level parallel programming languages. Several parallel Lisp and Prolog efforts, by far the most popular targets, have been underway in various universities and research organizations for some time. The need for high-level languages cannot be stressed enough. Even with a sophisticated tool set, such as the monitor-based tools described in Lusk[32], procedural programming (e.g., in C) is difficult and error-prone. The family of parallel logic programming languages derived from Prolog, and Prolog itself, offer much higher-level programming, protecting the programmer from the machine and reallocating the job of extracting parallelism and synchronizing parallel processes from the programmer to the system.

Note that although the above goals of high-level parallel language programming are often touted, the realities of implementation may cause the designers to mislay their initial goals. Of course there is no point to designing a parallel system that cannot achieve speedup. However, more strongly, as Lusk et. al. [33] claim:

> "The bottom line for evaluating a parallel system is whether it is truly competitive with the best sequential systems. To achieve competitiveness, it is necessary to make a parallel logic programming system with a single processor execution speed as close as possible to state-of-the-art sequential ... systems, while allowing multiple processors to exploit parallelism with the minimum of overhead."

Of the various sources of parallelism present in logic programs [17] AND- and OR-parallelism (or their combination) offer special promise and are currently being considered in several proposed parallel logic programming systems (e.g., [48, 3, 16, 66, 33, 20, 28]). Efficient techniques for implementing OR-parallelism have been proposed and are currently under development by various groups. AND-parallelism, although offering advantages such as being able to exploit parallelism in determinate programs and inherent efficiency, has until recently been difficult to implement due to the overhead involved in handling shared variable bindings and because of its interaction with "don't know" non-determinism. Consequently, many proposed parallel logic programming systems that exploit this type of parallelism do not implement the conventional "don't know" non-deterministic semantics of logic programs [31] and implement *committed-choice* (i.e., "don't care") non-determinism instead [48].

This study compares the design and execution performance of two parallel logic programming architectures, both of which have been implemented by independent groups [45, 33] on Sequent shared-memory multiprocessors[47]. Aurora is an OR-parallel Prolog system retaining the full semantics of Horn Clause logic (i.e., backtracking non-determinism). KL1 is an AND-

parallel FGHC system that is a committed-choice architecture (i.e., no backtracking). Although the performance measurements presented here do not compare favorably with C—our feeling is that in their current forms, these systems execute at least 10-50 times slower than equivalent programs written in C—we doubt very much that large parallel programs can be written in C with the same ease. Compiler technology is expected to bridge some of this gap, i.e., current logic programming compilers still lag behind procedural language compilers.

A high-performance programming system enables the development of powerful (parallel, memory efficient, declarative, fast) algorithms, as well as the efficient execution of the architecture. The former without the latter results in a top-heavy system, e.g., GHC as compared to FGHC. In this case, the language is too complex to implement efficiently. The latter without the former results in the opposite: a language easily implemented, but inherently weak, e.g., FGHC as compared to Prolog. Note that the examples of GHC, FGHC and Prolog given above are opinions not just of the author, but of the designers of the languages themselves. It has been said that committed-choice languages are just "machine languages" with which to build more complex languages. A potential pitfall of this approach is the loss of efficiency due to levels of meta-interpretation and/or translation. This is an old argument about "semantic gap" [36], i.e., that the user language and machine architecture should be as "close" as possible for efficient execution. Within the past ten years however, research in reduced instruction set computers (RISC's) has shown that semantic gap can be closed quite effectively by optimizing compilers, instead of powerful architectures. It remains an open question as to whether high-level languages can be efficiently implemented on top of committed-choice languages in any manner. On the other hand, it should be noted that Prolog may fall prey to the "top-heavy" problem stated above. The Aurora system measured here is the initial stage of a more general AND-OR-parallel system called Andorra [4]. The overheads of exploiting both types of parallelism may negate much of the gain. It is relatively clear, however, from the results of this study, that full unification, backtracking and dependent AND-parallel (stream communication) synchronization are all necessary in a high performance logic programming system.

Architectures are compared in this study at various levels of abstraction, in an effort to give successively refined models of performance. At the top level, raw execution timings of parallel emulators running on a host multiprocessor are presented. Speedups measured on various benchmarks are compared and analyzed. At the next level, dynamic architecture execution characteristics are presented, such as memory references made and procedure calls executed. These measurements give more insight into the algorithm differences of the benchmarks and power of the languages and architectures. At the third level, cache referencing characteristics are presented, such as miss ratio and traffic ratio. These measurements, collected on instrumented versions of the emulators, help understand the strengths and weaknesses of the storage models

of the architectures. From all three levels of analysis combined, a picture emerges of how OR- and AND-parallel systems, as well as noncommitted-choice and committed-choice languages compare in terms of of program performance.

The paper is organized as follows. The Prolog and FGHC languages are briefly reviewed in Section 2. The corresponding Aurora and KL1 architectures are described in Section 3. In Section 4, a review of the relevant published literature is given. In Section 5, the methodology of this study is outlined, including detailed descriptions of the parallel cache simulator and the benchmarks measured. In Section 6 the Aurora and KL1 architectures are discussed once again, this time with respect to the measurement tools previously described. Whereas Section 4 describes the architectures *as they were designed*, Section 6 describes the architectures *as they were modeled*. Sections 7–9 present the main body of this study: the statistical measurements of memory performance. Finally, in Sections 10 11, conclusions are given and future research plans are outlined.

2 Languages

In this section, two parallel logic programming languages are introduced at a fairly elementary level. The languages, Prolog and FGHC, can be viewed as representing a far larger family of languages based on their paradigms. Prolog is the language base of the Aurora [33], ANDORRA [4], PEPsys [66, 11], and RAP [28] systems, to name a few. FGHC is closely related to Flat Parlog [13, 14] and (somewhat less related to) Flat Concurrent Prolog [48].

For a good introductory book about Prolog please refer to Sterling and Shapiro [50], Bratko [5], and Clocksin and Mellish [15]. Maier and Warren [34] is a good introduction to the implementation architecture of Prolog. Gregory [27] gives a coherent introduction to committed-choice languages, specifically Parlog.

2.1 OR-Parallel Prolog

The OR-parallel Prolog language discussed in this paper is essentially SICStus Prolog [9] with user annotations to denote procedures that are permitted to be executed in OR-parallel search. Prolog is a logic programming language based on Horn clauses of the form:

$$H :- B_1, B_2, \ldots, B_n.$$

where H is the head of the clause and the goals B_i comprise the body of the clause. The head and goals may each contain zero or more arguments. Arguments are *terms*, for example variables, integers, atoms, or complex terms such as lists and structures (which may be nested). A variable is an unbound value cell which is defined within the scope of the clause only (c.f.,

there are no free variables as in Lisp, there are no global or non-local variables as in Pascal). Note that head arguments correspond to formal parameters in a procedural language, and body goal arguments correspond to passed parameters.

Procedures are composed of sets of one or more clauses with the same name and argument number (arity). A procedure is non-determinate if more than one clause can successfully execute for a given set of arguments. A Prolog procedure call involves *unifying* the goal (the caller) with a clause head (the callee). If no clause head can unify, the call *fails*. Failure returns control not necessarily to the caller, but to the last *choice point*, i.e., non-determinate procedure with alternative clauses (this is called *backtracking*). Prolog is applicative in that a variable can be bound at most once within scope with determinate goals, but if the scope contains non-determinate goals, backtracking can reset the binding of a variable.

SICStus OR-parallel Prolog allows the user to annotate any procedure with a `parallel` declaration. If the procedure is non-determinate, such annotation permits the system to fork independent processes for each alternative clause at the procedure's choice point—this is called a *branch point*. One can envision the creation of a "process tree" consisting of nodes and arcs. A node with more than one branch is a branch point. Nodes with only one branch may or may not be branch points—the scheduler may decide to execute potentially parallel code sequentially because of scheduling heuristics. Note that solutions to the program are found at the leaves of the tree. A description of this abstract model is given by Warren[64] and reviewed in Section 4.2.

As a trivial example of Prolog programming, two procedures are shown in Figure 1. The important point of this example is that `gen` is determinate,[1] generating a list of integers from `0` to `N`, whereas `del` is nondeterminate, generating multiple solutions. For example,

```
?- del([1,2,3],X,T)
```

produces the answers:

```
X=1, T=[2,3]
X=2, T=[1,3]
X=3, T=[1,2]
```

The `unroll` procedure is an unrolled version of `del` that can spawn four children per branch point. The benchmarks presented in this study typically use procedures like `del` and `unroll` to exploit *all* OR-parallelism in the program.

[1]The special builtin predicate `!/0` (called cut) is used to remove all alternatives up to and including the procedure in which it lexically appears. In other words, any and all choice points, including the choice point (possibly) created for the procedure containing the cut, are removed. Cut is primarily used, in a somewhat unclean programming style, to obviate the need for checks (guards) in alternative cases. Cut also causes immediate removal of choice points, thus increasing the efficiency of storage management.

```
gen(0, []) :- !.
gen(N, [N|X]) :- M is N-1, gen(M,X).

:- parallel del/3, unroll/3.

del([X|T], X, T).
del([H|T], X, [H|R]) :- del(T, X, R).

unroll([X|T], X, T).
unroll([A,X|T], X, [A|T]).
unroll([A,B,X|T], X, [A,B|T]).
unroll([H|T], X, [H|R]) :- unroll(T, X, R).
```

Figure 1: Examples of Prolog Procedures

2.2 AND-Parallel FGHC

Flat Guarded Horn Clauses (FGHC) [58] is also a language based on Horn clauses. An FGHC clause is of the form:

$$H :- G_1, ...G_m | B_1, B_2, ... B_n.$$

where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. In FGHC, as in Prolog, procedures are composed of sets of clauses with the same name and arity. Unlike Prolog, there are no non-determinate procedures. Execution proceeds, like Prolog, by attempting unification between a goal (the caller) and a clause head (the callee). If unification succeeds, execution of the guard goals are attempted. In FGHC, these goals can only be system-defined builtin procedures, e.g., arithmetic comparison. If the guard succeeds, the procedure call "commits" to that clause, i.e., any other possibly good candidate clauses are dismissed. If the head or guard fails, another candidate clause in the procedure is attempted (if all clauses fail, the program fails). In FGHC there is a third possibility however: that the call *suspends*. This is described in detail below.

FGHC restricts unification in the head and guard (the "passive part" of the clause) to be input unification only, i.e., bindings are not exported. Output unification can be performed only in the body part (the "active part"). These restrictions allow AND-parallel execution of body goals and even OR-parallel execution of passive parts during a procedure call (the implementation discussed herein executes passive parts sequentially and executes body goals in a depth-first manner). Synchronization between processes is inherently performed by the requirement that no output bindings can be made in the passive part. If a binding is attempted, the call *potentially* suspends. If none of the clauses succeeds, and one or more potentially suspend, then the procedure call suspends (possibly on multiple variables).[2]

[2]In the dialect of FGHC used in this study, there is a special clause called **otherwise**. Any number of **otherwise** clauses may appear in a procedure, each appearing as if a unit clause, but actually belonging to the

```
gen(N, X) :- N=:=0 | X = [].
gen(N, X) :- N>0 | X = [N|Xs], M := N-1, gen(M,Xs).

append([A|X],Y,Z):- true | Z=[A|Z1], append(X,Y,Z1).
append([],Y,Z):- true | Z=Y.

merge([X|Xs], Ys, Z) :- true | Z = [X|Zs], merge(Xs, Ys, Zs).
merge(Xs, [Y|Ys], Z) :- true | Z = [Y|Zs], merge(Xs, Ys, Zs).
merge([], Y, Z) :- true | Y=Z.
merge(X, [], Z) :- true | X=Z.
```

Figure 2: Examples of FGHC Procedures

When any of the variables to which an export binding was attempted are in fact bound (by another process), the suspended call is resumed. These semantics permit stream AND-parallel execution of the program, i.e., incomplete lists of data can be streamed from one parallel process to another in a producer/consumer relationship. For example, when a stream runs dry, the consumer receives the unbound tail of a list and suspends. When the producer generates more data, the consumer is resumed and continues processing the transmitted data. In the implementation discussed herein, these data structures all reside in shared memory.

The FGHC abstract execution model is a reduction mechanism wherein the initial user query (a set of goals) is reduced to the empty set. A single goal is reduced by unifying it successfully with a clause and then replacing the goal with the body goals of the matching clause. Reductions of goals can proceed in any order. Superimposed on this model is a suspension mechanism that causes goals to suspend and resume. A "process tree" model could be developed for FGHC as in OR-parallel Prolog, but has not been because it is less useful (the main backbone of the Prolog tree are branch points, of which there are none in FGHC). Such a model might be useful however for scheduling, to help determine the granularity of goals. The FGHC architecture studied here uses a simplistic "pool of goals" model [11].

As a trivial example of FGHC programming, three procedures are shown in Figure 2 that are used later in this paper. `gen` corresponds to the Prolog procedure in Figure 1. `append` is determinate list concatenation. `merge` nondeterminately joins two streams (the first two arguments) into one (the last argument). `merge` shorts itself when it receives a [] from either input stream. This procedure is useful for routing messages in an object-oriented programming style.

---

procedure where they lexically appear. If none of the clauses preceding an `otherwise` succeed, and one or more clauses can potentially suspend, then the procedure call suspends. Only if all the clauses preceding an `otherwise` fail, will the remainder of the clauses (up to the next otherwise) be checked. `otherwise` is primarily used, in a somewhat unclean programming style, to obviate the need for guards, thus speeding-up the program. Its effect can be significant.

3 Architectures

An architecture is an instruction set, storage model, and execution mechanism implementing a language. The OR-Parallel Prolog architecture (Aurora[64, 33]) and the FGHC architecture (KL1[25]) are called "high-level" architectures because their instruction sets are more abstract than those of conventional computers. Both of these architectures have been implemented on the same general-purpose host (the Sequent Balance and Symmetry multiprocessors[47]) via emulation. Although these implementations of both systems are preliminary and not of commercial quality, they represent two of a very recent group of *true-parallel, high-level language implementations.*

The architectures of these two systems are summarized in this section. Concentration is placed on those parts of the architecture that radically effect performance. Not all of these aspects come into play in the benchmarks studied here. For instance, although garbage collection (GC) is extremely important in both systems when running large applications, GC is not a significant performance factor here. Other aspects, such as compiler optimizations, are also extremely important, and unfortunately unequal between the two systems. A group of such differences affect system performance in unison. Thus separation of individual effects and system calibration are difficult. The complexity of these systems must be kept in mind when interpreting the results presented in later sections.

3.1 Overview

3.2 Engine Architecture

The instruction set design of an architecture determines the instruction execution times, the memory bandwidth required, and the compiler optimizations allowed. The Aurora system uses Carlsson's (SICStus) version of the Warren Abstract Machine (WAM)[61] instruction set. Modifications were made to implement binding, dereferencing, and trailing with respect to binding arrays (see Section 3.3). The KL1 system uses Kimura's version of the WAM, called KL1-B[30]. KL1-B is both simpler than the WAM because backtracking has been removed, but also more complex than the WAM because both suspension and locking mechanisms have been integrated. Both systems use the compilation technique of clause indexing.

The compilers of the two systems differ in sophistication. The Aurora compiler generates optimized code for shallow backtracking, i.e., backtracking among the clauses of a procedure. The KL1 compiler generates somewhat round-about code with redundancies in order to reduce the locking interval on a variable being bound (c.f., Foster's compiler for Flat Parlog[21]). Maybe the greatest difference is in the instruction formats: Aurora uses a large set of "fused" WAM

instructions, many are combinations of two or three of the original instructions as defined by Warren. The KL1 compiler used in this study has few fused instructions (indexing instructions may be considered fused, since they are more sophisticated than those in the WAM). Thus KL1 programs tend to execute many more instructions than Aurora programs (interestingly, KL1 code locality is much *higher* than Aurora code locality, as discussed in Section 8.2).

3.3 Binding Mechanism

In parallel systems, bindings are the means by which processes communicate among themselves and with the outside world. In Aurora, parallel processes executing a non-determinate procedure produce independent solutions, i.e., they can potentially produce conflicting, but valid, bindings. To implement multiple bindings, the Aurora system uses a *binding array* per processor wherein bindings to variables shared among branches reside (i.e., bindings to variables that may potentially differ among the processors). In fact, Aurora implements two types of binding arrays: *local* and *global*. The local array is used for variables on the environment stack, and the global array is used for variables on the heap. Both areas are local to the PE, i.e., are not shared by other PEs. As discussed in Warren [64], binding arrays keep dereferencing and (un)binding operations constant time operations. However, binding arrays impact task-switching time because the overhead of "spawning a process" is the work required changing the values in the binding array to reflect the new process's location (in the process tree).

The binding array is a stack of values, the growth of which follows the movement of a worker around the OR-tree. An *unconditional binding*, i.e., a binding to a non-shared variable, need not use the binding array mechanism and is performed *directly* on the variable cell itself. If the binding is *conditional*, i.e., "if there is a branchpoint at or below the point the variable is created and above the point at which it is bound"[64], it is trailed (both variable cell address and value) and the bound value is written to the binding array cell, *not* to the variable cell itself. Initially, a variable cell points into the binding array with an "unbound" tag. During conditional binding, the variable cell remains pointing to the array, and only the array cell is modified. Dereferencing in Aurora is therefore straightforward: if the tag is "unbound," then the corresponding binding array cell is dereferenced. Of course, each worker[3] has its own array and all array's have a one-to-one correspondence for each variable encountered.

During failure (upon backtracking), the trail is popped in order to unbind all spurious bindings. The trail address entry points to the variable, which if conditional, points to the binding array. The trail entry also holds the value of the binding. This is used during task-

[3]The terms *worker*, *process*, *processor*, and *PE* are used interchangeably and informally in this paper. Both the Aurora and KL1 systems studied here allocate a single process per processor. In the Aurora literature, this is called a worker.

switching as follows. When an idle worker moves *up* the OR-tree, it *de-installs* bindings from the trail in the same manner as if it *failed* back to destination node. When an idle worker moves *down* the OR-tree, it *installs* bindings from the trail. The portion of the trail delimited by the start node and the destination node is read. For each trail entry read, the value must be installed in the worker's binding array. Note that OR-tree operations such as these must be protected by locks. In Aurora, workers move incrementally up and down the tree, locking and (de)installing binding for each node separately. Note that locking is *not* necessary for (un)binding or dereferencing variables.

In KL1, AND-parallel execution implies that all processes have equal authority to bind any variable at any time. Thus the binding problem becomes a locking problem. The binding (in the active part of a clause) of variables (passed from the passive part of the clause) must be locked. This is related to code generation because to reduce the locking penalty, somewhat roundabout code is generated to minimize locking times (as mentioned in Section 3.2).

Dereferencing in KL1 involves following a pointer chain to a value, possibly an "unbound" or "hooked" value cell. In some cases, safe dereferencing is necessary, i.e., the pointer chain must be locked as it is traversed in order to prevent another PE from racing toward binding it. This is implemented in a straightforward way by locking and then unlocking each pointer as it is traversed.

In all cases of KL1 dereferencing, the initial variable cell may be overwritten with its dereferenced value. Overwriting speeds-up subsequent dereferencing, but more importantly, reduces sharing of data among PEs. This optimization cannot always be performed in Prolog because of backtracking constraints. In both Prolog and KL1, dereferencing chains are very short, almost always immediate or single referenced data [56, 43]. In the case of sequential Prolog, the overwriting optimization does not pay off; however, in KL1 it reduces bus traffic by decreasing reads to shared data on the heap.

Binding a KL1 variable must always be protected by a lock to prevent another PE from also binding it. Variables are never trailed however because there is no backtracking.

3.4 Scheduler

The process scheduler must be efficient in two major respects. First, the work must be evenly distributed among the processors (good load balancing). Second, the overhead of process spawning/suspending/resuming must be low. If only large-granularity goals are spawned on different processors, both of these criteria will be met. Whereas goals are stored in a tree structure in Aurora, in KL1 all goals are treated equally, and stored in *goal-lists* local to each process. Both the Aurora and KL1 research groups have explored various scheduling mechanisms[49, 7, 8, 53]; however, the measurements presented in this paper were made on the

following fixed systems.

Aurora scheduling is performed locally by the process with a distributed "tree-walking" algorithm (the "Argonne scheduler" [7, 64]). An idle worker (one that succeeded or failed, and therefore has no further work) traverses the OR-tree, constrained by several heuristics, searching for work. This traversal is in the *public* section of the tree, above all the *private* sections where the busy workers are executing in their WAM engines. The separation of public and private sections is necessary to keep efficiency high by obviating the need for locking in the private section. Since idle workers cannot travel down into private branches, no locking is necessary and the WAM engines are as efficient as in sequential Prolog. In certain cases however, an idle worker communicates with a busy worker by raising a (soft) interrupt flag that is checked once per reduction in the engine.

The idle workers traverse the tree incrementally, i.e., locking one node at and time and (de)installing bindings for that node to regain consistency. Note that locking the node prevents other workers from walking by in either direction. Traversal must be kept to a minimum because although an idle worker has free CPU cycles, traversal causes cache interference and increases the bus bandwidth requirement. Therefore, the top-most node of a private section is periodically *released* to the public section. This implies that new work is created closest to where the idle workers are positioned.

An idle worker positioned at a public node must make a decision about what to do. All the information necessary to this decision is given in the current node and the nodes surrounding the current node. If the node has as yet unexecuted alternatives, the idle worker creates a branch for one and begins execution. If the node has no further alternatives, but one of its children does, the idle worker moves down to that child. If the node and its children have no further alternatives then the idle worker moves up, etc. There are several possibilities specified in this "move to work" logic, and completeness is guaranteed. Unexecuted alternatives may appear, however, at any time in the private section. Thus the default action of an idle worker, when all other options look pointless, is to sleep for a bit (see Section 6.2.3) and then try to make a decision again.

The Argonne scheduler performs quite well for programs with an abundance of OR-parallelism, as reported by Butler et. al. [7] and in this report also. However, as shown here, when parallelism is scarce, the scheduler tends to eat up many bus cycles in the "move to work" loop described above. The fundamental problem is how to automatically regulate the number of active workers, e.g., running on two PEs in one portion of a program, and then eight PEs in another. Shen and Warren also point this out from higher-level simulations[49]. The problem appears to be much worse than they anticipated however, affecting even small numbers of PEs.

In KL1,[4] scheduling is performed in a semi-distributed manner. An idle process (one with an empty goal-list) requests work from a busy process, via a (soft) interrupt. The interrupt flag is checked once per reduction. A pointer to a goal available to be executed is passed back to the idle PE. Thus the goal-lists are virtually independent, but in actuality, become intertwined as execution proceeds.

Scheduling an idle KL1 worker has none of Aurora's overheads of (de)installing bindings, locking nodes, making private nodes into public nodes, etc. Any worker can execute any goal at any time. The problem in KL1 is one of granularity: it is not efficient to give an idle worker a trivial goal to execute because the goal will be quickly completed and then the idle worker must issue another interrupt.

An idea of compile-time granularity analysis was developed for KL1 wherein *weights* are calculated as estimators of the relative granularity of procedures in an FGHC program[57]. The idea is based on the scheduling heuristics used in the Argonne scheduler. Preliminary experiments show that the method does not benefit FGHC as much as Prolog because FGHC often has critical timing dependencies that can incur large synchronization overheads if tampered with. In fact, it appears that for committed-choice languages on shared memory multiprocessors, reducing suspension overheads is more important for performance than improving scheduling.

3.5 Storage Model

The critical issue however, is the ability of WAM's parallel offspring (Aurora and KL1) to retain an efficient storage model. All parallel computer architectures execute on some organization of processors coupled with memories through an interconnection network. Because the memories necessary to hold the working set of large application programs are not large enough to be integrated with the processors, and because processors need to communicate (to varying degrees) to execute a program in unison, memory/network bandwidth inevitably becomes a bottleneck to performance. Thus the exploitation of locality, both spatial and temporal, becomes critical to the architecture.

Memory management is important to retain the spatial locality needed to make efficient use of local caches. In addition, efficient memory management creates less garbage and therefore garbage collection is incurred less often. In Aurora, a group of intertwined stacks (called a *stack-group*) is assigned to each PE. An Aurora stack-group is similar to that of the WAM, containing a control stack (choice-points), local stack (environments), global stack (heap), trail, and binding array(s). The stacks are the physical storage areas comprising the virtual OR-tree. Consider branch-points (nodes) as the most obvious case. The nodes of the OR-tree

[4]Again, this description applies to the system measured in this study — other KL1 systems may differ.

are "flattened" into a set of control stacks, one per PE. In other words these "cactus" stacks logically form the OR-tree. The other stack types in the group are related to the control stack in the same manner as in the sequential WAM.

One difference between Aurora and WAM is the potential creation of holes or *ghost nodes*[64, 28]. Holes may form in the stacks when a parent stack spawns a child stack, and the child then spawns a grandchild *on the parent's stack*. The frequency of this type of garbage is not currently known. Aurora can recover this garbage when/if natural backtracking reclaims stack space around the hole. Another difference is the size of stack frames in Aurora and the WAM. Aurora branch-points are larger than WAM choice-points (contain six additional entries), containing information necessary to manage OR-parallel branches. As previously mentioned, the trail is also bigger with double word entries.

The Aurora local and global stacks are accessed in much the same way as WAM. During execution in the private part of the OR-tree, these stacks and the trail are accessed in a WAM-like manner, offering high locality. In addition, the binding arrays are also accessed, most likely in a more random manner. Note however that the binding arrays are purely local to the PE and are not shared. The local and global stacks and trail, although used locally, may be read from other PEs (e.g., during dereferencing). During a task switch, the top of the local and global stacks are allocated to a different branch of execution in the OR-tree. Thus locality is somewhat lessened.

The Aurora control stack is accessed in a more random fashion because when a worker becomes idle, it must search the control stacks for work. This occurs in the public portion of the OR-tree, and involves complex scheduling heuristics to determine where the idle worker should search. In general, control stack referencing is expected to have little locality and a high degree of sharing among PEs. In addition, as the worker traverses the tree, the trail is used in an equally disjoint manner to (de)install bindings in the binding array(s).

In KL1, each processor has a storage group consisting of a heap, goal record list, suspension record list, and communication area. The lists are allocated from a larger group of free-lists, split among the processors to avoid contention. The heap is used to store all values, atomic and structures. A goal record corresponds to an environment in the WAM; however, all bindings are made to the heap to facilitate deallocation of the goal record. A suspension record is a far simpler (two word) structure necessary to manage synchronization. When output unification is attempted in the head of a clause, the variable in question is pushed onto a *suspension stack* and the next clause is attempted. If none of the clauses of the procedure succeed, then the procedure call is officially suspended. The suspension stack is popped and each variable is made to point ("hooked") to a newly created suspension record. The suspension record points to the suspended goal (procedure call). When/if the variable is bound, the hooked goal is resumed.

The suspension stack is not considered a major storage area in the architecture because it rarely grows large (just a few entries—it is similar in status to the unification stack or PDL of the WAM). Finally, the communication area is used to pass messages from an idle PE to a working PE, requesting work.

Goal records are accessed for the most part in a single-write, single-read manner. This corresponds to Warren's "goal-stacking model" for Prolog [62]. A goal is reduced to a clause body which replaces it in the goal list. Thus each goal is actually written and read just once, and not kept for future reference like a Prolog environment. The goal list is accessed in a first-in-last-out (FILO) manner. Locality is thus high and sharing low except if spawning is frequent.

When a goal is spawned, the goal is simply rewired from one PE's goal list to another PE's goal list. This saves copying, and on a shared memory organization is the lowest cost method of task switching. However, this method implies that if goal spawning is frequent, spatial locality is destroyed. This is similar to the problem in Aurora: if task spawning is frequent, i.e., OR-parallel goals have too fine granularity, then many child nodes must be created at great cost with almost no computational benefit. Thus Shen suggested a threshold heuristic (see Section 4.2). In the KL1 architecture such a heuristic is not used (although related ideas have been examined by the author [57]). In any case, Aurora is different in that spawning a task involves creating a new branch, and that branch physically resides in the stack-group of the worker. In KL1, the spawned goal physically resides wherever the goal record was allocated from a free-list.

The KL1 heap is accessed as in Prolog; however, there is no backtracking to automatically reclaim heap space. Thus the heap referencing marches monotonically through the allocated area until garbage collection occurs. The suspension area is randomly accessed, but frequency of access should be low in most programs. The communication area is accessed in a single-write, single-read manner. The messages are sent mainly when seeking work and when resuming a goal on another PE.

In general, KL1 storage management is simpler than Aurora's, but the KL1 model creates garbage at a significantly faster rate, as discussed below.

3.6 Garbage Collection

All languages that dynamically create structures require some form of garbage collection (GC). In Aurora, the WAM automatically recovers memory upon backtracking, i.e., when searching for all solutions to a non-determinate problem, memory used to explore bad paths is easily recovered. However, the determinate portions of programs can still produce garbage (in the form of temporary data structures needed to get from one intermediate point to another,

and then discarded). There is no explicit GC for determinate garbage in the Aurora system measured in this study.

FGHC generates more garbage than does Prolog because OR-parallel search is unwittingly *simulated* by the architecture[59] which therefore cannot automatically recover memory upon backtracking. Taking another view, because there is no backtracking, logical unification is incomplete, i.e., it cannot be undone (via the trail). Thus the construction of a solution to a problem in FGHC must frequently *copy* data structures. Prolog, on the other hand, can avoid copying by rebinding the same logical variable many times. With shared logical variables, certain algorithms are extremely efficient in memory usage and execute time (e.g., register allocation or resolving labels in compiled code [60, 41] and constraint problems -see **HKqueens** and **Puzzle** in this study).

Various methods of GC are currently being explored by ICOT[37, 26, 12]. These methods are beyond the scope of this paper and will not be discussed. The KL1 system described in this paper uses sequential "stop and copy" GC only. However, for the benchmarks studied, GC is not a significant factor.

4 Literature Review

In this section, the published literature related to this study is examined. These papers fall into two approximate categories: the work done at ICOT on performance measurement of stream AND-parallel architectures, and the work done at Manchester University and Argonne National Laboratories on performance measurement of OR-parallel architectures.

There are many other papers available related to committed-choice architectures (e.g., FCP and Flat Parlog abstract machines) and non-committed-choice architectures (e.g., RAP and PEPsys abstract machines). However, the two architectures chosen for this study, KL1 and Aurora, alone are reviewed. Many of the research results concerning these two architectures can be directly related to the other models.

4.1 KL1 Research Papers

Matsumoto [35] characterizes the behavior of a coherent cache design specialized for KL1 execution. He measured one benchmark, **BUP** (Bottom-Up Parser) executing unrealistic and small input data. He used a "pseudo-parallel" KL1 emulator to produce an address-trace file for later input to a cache simulator. The emulator round-robin scheduled processes, switching each reduction. Thus the measurements do not accurately reflect real locking behavior. Matsumoto's primary result is that the cache optimization of a "direct write" operation (that avoids fetching a block from memory, for example to be used when creating a structure on

the top of the heap), saves 31% of the total required bus bandwidth of the program. Similar optimizations (read-purge and read-buffer cache operations) for the goal and communication areas saved an additional 6% of the bus cycles. Thus using direct write for the heap alone offered 84% of all savings, and therefore in this study, for KL1, direct write is used only for the heap. As is discussed in Section 9.1, the real parallel simulator used for this study gives more accurate timing and shows lower suspensions for **BUP** than measured by Matsumoto. The primary effect of this reduction is to increase the relative weight of goal and communication traffic, and the relative importance of the read-purge and read-buffer optimizations.

Matsumoto's paper is a good introduction to the cache protocol measured in this study also. He discusses tradeoffs in cache organization, such as number of sets and line size. These issues are not investigated in this study, rather we defer to Matsumoto's suggestions of four sets and four word lines.

Nishida [38] presents measurements and analysis of the *multiple reference bit* (MRB) incremental garbage collection method [12]. He measured the **BUP** benchmark previously mentioned[5] executing on another "psuedo-parallel" emulator developed specifically for MRB studies. MRB garbage collection (MRB-GC) concerns the heap only (the goal and communication areas can be incrementally reclaimed as noted by Matsumoto[35]). Nishida's main result is that MRB-GC reduces heap bus traffic significantly for a few PEs, and then loses its ability with increasing numbers of PEs. The reason given is that MRB-GC causes cache blocks to be shared, thereby increasing the frequency of cache-to-cache invalidations with increasing PEs. On eight PEs, Nishida's data indicates that MRB-GC reduces heap bus traffic by about 60%. Scaling this savings by the expected percentage the heap contributes to bus traffic in **BUP**, 26–44% (see Section 9.1), we get a savings of 15–26%. On 12 PEs, the effect of invalidations becomes pronounced, and the savings decreases to about 7–12%. In all these cases, the traffic savings is significant. Unfortunately, MRB was not implemented in the parallel emulator used in this study, and therefore comparisons with Nishida's work cannot be undertaken.

Taki [53] presents measurements of two **8-Queens** (FGHC) programs running on the Multi-PSI V1 multiprocessor. The purpose of his study is to analyze inter-PE communication costs on a *distributed* KL1 multiprocessor. The benchmarks incorporate user-defined *pragma* to allocate the goals to specific PEs (varied from 1–6). The paper is an interesting introduction to the problems involved in the communicating clusters of the PIM [25]. However, no results are given estimating the performance or communication costs of real application programs.

Sato [44, 45] describes the parallel KL1 emulator (also used in this study) and presents measurements of its execution of a set of benchmarks. The papers present a good overview of

[5]The **BUP** program measured by Matsumoto finds all solutions for parsing a single complex sentence. The **BUP** program measured by Nishida was modified to parse ten independent sentences concurrently.

the KL1 architecture, its instruction set and storage model. In [44], two distribution methods are compared: random (upon procedure call, the called goal will automatically be thrown to a random PE) and on-demand (an idle PE will ask for a goal to execute). Both schemes use pragma, although defined differently than Taki (above). Sato's main result is that on-demand distribution is better than random distribution for two benchmarks: **8-Queens** and **BUP**. Maybe more interesting is the difference between the two benchmarks given on-demand distribution. More realistic in modeling real applications than **Queens**, **BUP**'s percentage idle time is 17 times larger than **Queens**'. **BUP**'s distribution ratio (percentage goals thrown to other PEs) is 31 times **Queens**'. These differences indicate that any parallel performance measurements of **Queens** will be misleading at best.

Sato [45] extends his measurements to include **Quick-sort**, **Prime**, and **Maxflow**[52]. These additional benchmarks have little speedup (using the Sequent Balance multiprocessor, a speedup of 8 on 16 PEs). Sato's main result is that the most important factor degrading system performance is idle time, followed by number of suspensions. Locking and inter-PE communication have minimal effect.[6] The result that idle time is most critical to program performance implies that the KL1 system has low overhead in exploiting parallelism. This also restates two tautologies. *Programs with little parallelism get poor speedup. Programs with sufficient parallelism require a fair and efficient load distribution method.* Within the benchmark suite Sato measured, the distribution ratio varied from 1.7% to 6.7%, a factor of four. The suspension ratio (suspensions per reduction) varied from 0.0 to 0.4. Thus the expected ratios of real application programs are unknown. In addition, the program with the least speedup, **Maxflow**, had one of the highest distribution and suspension ratios. This implies that the on-demand distribution is not efficient for **Maxflow**, and as Sato points out, more efficient scheduling mechanisms must be studied. In the the study presented here, suspension ratios vary from 0.0 to 0.09.

A collection of FGHC programs is given in Takagi [52]. Of the 16 programs, limited evaluation measurements are give for three of them. None of the evaluation was done on a parallel system. One of the programs, **Pascal**, is used in a modified form as a benchmark in this study.

4.2 Aurora Research Papers

An introduction to OR-parallel computation and the Aurora system in particular are given by Warren [63, 64] and Lusk et. al.[33]. Warren [63] discusses alternative designs for OR-parallel execution of Prolog. He analyzes several schemes: the "Argonne model" [6, 40] utilizing a "favored binding" optimization and hash binding tables, the "SRI model" utilizing binding

[6] Note that Sato [45] measured inter-cluster communication, in contrast to Taki's measurements [53] of *intra*-cluster communication.

arrays, and various other models. The conclusion reached is that perhaps an "SRI-Argonne model" is best; however, no hard data is presented. Since that time, Shen and Warren [49] and Disz et. al. [19] did extensive measurements and found that the "favored binding" optimization was not particularly effective. Therefore, later designs [64, 33], are based on simple binding arrays only.

Shen and Warren[49] present measurements and analysis of the Argonne model. They simulated the execution of 20 benchmarks, all small except for **CHAT** [65]. A psuedo-parallel simulator was used, where the time step was one reduction. The maximum size benchmark studied was 3662 reductions (c.f., the *minimum* size OR-parallel Prolog benchmark studied here is 33,595 reductions). Shen draws many interesting conclusions about OR-parallel execution, that have since steered the design of Aurora. There was limited OR-parallelism in the benchmarks studied, suggesting that limiting the number of PEs was most cost-effective. The "favored binding" optimization was found to be inefficient and therefore Aurora did not adopt the idea. Work distribution strategies were briefly examined and the scheme of spawning the highest choice point[7] in the OR-tree was found to match a simple method of spawning the first choice point created (FIFO). Aurora chose a variation of the former strategy. Shen also suggests placing a constraint on spawning choice points whereby a threshold number of reductions must first be made before the choice in enabled to spawn. The threshold attempts to discriminate between long and short branches eminating from the choice point. Aurora adopted this idea.

Disz et. al.[19] present timing and high-level measurements of OR-parallel Prolog benchmarks, measured on a real-parallel implementation of the Argonne model. Two of benchmarks studied are too small to use for cache studies. Another, **Semigroup**, is large enough and is analyzed here. Disz discusses the "favored binding" optimization in detail and analyzes its performance. In addition, the paper concludes that neither OR-parallelism or independent AND-parallelism [18], by itself, is sufficient for high performance systems. This conclusion is reenforced by a conclusion here that neither OR-parallelism or dependent AND-parallelism, by itself, is sufficient.

Warren [64] gives a more abstract view of OR-parallel computation in terms of an OR-tree. The nodes of the tree correspond to a task (a set of goals, clauses and bindings) that needs to be reduced. A node is reduced ("extended") into a new node below it where one of the goals is replaced by the body of a clause which it matches. If multiple matching clauses exist, a node may have multiple children. This type of node is called a branch-point and corresponds to sequential Prolog's choice-point. The execution of an OR-parallel program consists of extending the root task (the user query) until all branches in the tree are generated. Branches with a leaf

[7] By "spawning a choice point" we mean allowing an idle worker to execute an alternative branch from the choice point.

node containing an empty goal list represent solutions. Of course, multiple solution branches may exist.

Warren describes optimized operations on the abstract tree to manage its size. These operations, "dieback," "contraction," and "straightening," have correspondences with sequential Prolog's backtracking, WAM's `trust`, and `cut`, respectively.[8] In any case, the key point about the tree is that descendant nodes of an ancestor can share (in a read-only fashion) all variables (and structures) inherited from above. This is the basic idea behind binding arrays (see Section 3.3 of this paper). Warren's paper [64] can be considered a blueprint for the Aurora system.

Lusk et. al. [33] present a summary of the Aurora OR-parallel Prolog system. This paper is basically an updated version of [64] including some preliminary timing and high-level measurements. Five benchmarks (**Queens**, **Salt & Mustard**, **CHAT**, and **Tina**) were measured. Speedups of up to 14 on 16 PEs (Encore Multimax) are shown. It is noted that Aurora is 25% slower than SICStus Prolog from which it is derived, which is in turn twice as slow as Quintus Prolog[1]. These factors, in addition to the difficulty multiprocessors are having keeping pace with sequential microprocessors, are stated as the reasons that "truly competitive bottom-line performance" is not yet in sight. If however these results are compared to the published KL1 results, Aurora better achieves this goal. One of the purposes of this study is determine why Aurora is better achieving this goal. Is it because of more advanced implementation technology, fundamentally lower parallel overheads, or greater "semantic potency" than KL1?

Butler et. al. [7] present a summary of the "Argonne scheduler" used in the Aurora system (this is one of two alternatives currently implemented. The other is the "Manchester scheduler" written by A. Calderwood). Butler's paper is primarily concerned with the ramifications of implementing full Prolog in OR-parallel---specifically how to finesse side-effects by scheduling around them (note that benchmarks with side-effects are not studied here). The Argonne scheduler works in conjunction with a WAM engine for each worker (PE) in the system. At any point in time the worker is either busy (in the engine) or idle (searching for work or resting in the scheduler). The details of the scheduler algorithm are given by Butler and reviewed in Section 3.4. Butler shows relative speedups of 7.1, 6.6, and 7.8 on a Sequent Balance for the **Salt & Mustard**, **Zebra** and **Turtles** benchmarks respectively. However, also given is an example of a degenerate program that gets little speedup because of incompatible scheduling protocols. That example and two examples given in this study indicate that the Argonne scheduler is more sensitive to programs that do not suit it, than is the simple KL1 scheduler.

In as yet unpublished works, Calderwood [8] and Szeredi[51] present a great wealth of high-level data measured from a large group of Aurora benchmarks. Calderwood analyzes the performance of his own Manchester scheduler, in comparison to the Argonne scheduler.

[8]The Aurora system measured in this study performs dieback and contraction but not straightening.

4.3 Summary

The various papers reviewed in this section are at the very edge of a new field of research in parallel logic programming. It is therefore not surprising that the papers show little rigor in the performance analysis given. Relative speedup may be used to illuminate an architecture in a friendly light. Inefficient, but highly parallel, benchmark programs may be used to illustrate efficient scheduling. Few papers make any realistic comparisons between systems (Foster and Taylor [21] is one exception).

The purpose of the research study presented here is to correct some of these deficits. A multi-level performance analysis is given of both dependent-AND and (independent) OR parallel logic programming systems. The architectures are compared empirically at both high (e.g., number of reductions) and very low (e.g., bus traffic ratio) levels, *for the same benchmark programs.* Although this study fails in several respects—most notably in that large application programs could not be measured—it is hoped that this paper encourages members of the logic programming community to fairly, quantitatively, and accurately access the value of their systems.

5 Methodology

The Aurora and KL1 system architectures were measured and analyzed empirically by studying the results of executing a set of benchmark programs. The benchmark programs were collected and written to solve a given set of problems in both Prolog and FGHC. In most cases, a group of programs were written for a given problem, and compared for their speed. Through this process of refinement, the benchmark programs presented represent well-written relatively efficient programs.

The benchmarks were translated by compilers for their respective languages, and the resulting object files were executed by abstract machine emulators. These emulators run on a Sequent Symmetry multiprocessor and are truly parallel. These tools are illustrated in Figure 3. The basic Or-parallel Prolog emulator is the Aurora system written by various researchers at the Swedish Institute of Computer Science (SICS), Manchester University and Argonne National Laboratories (ANL). The Aurora compiler [9] was written by M. Carlsson of SICS. The basic FGHC emulator is the KL1 system written by M. Sato of ICOT. The KL1 compiler was written by Y. Kimura of ICOT. In Section 7 it is shown how the compilers compare in terms of quality of code produced, and how the emulators compare in terms of execution efficiency. Overall the two systems are closely calibrated and therefore allow a fair comparison of both raw timings and instrumented simulations.

Each architecture is emulated at three different levels of abstraction:

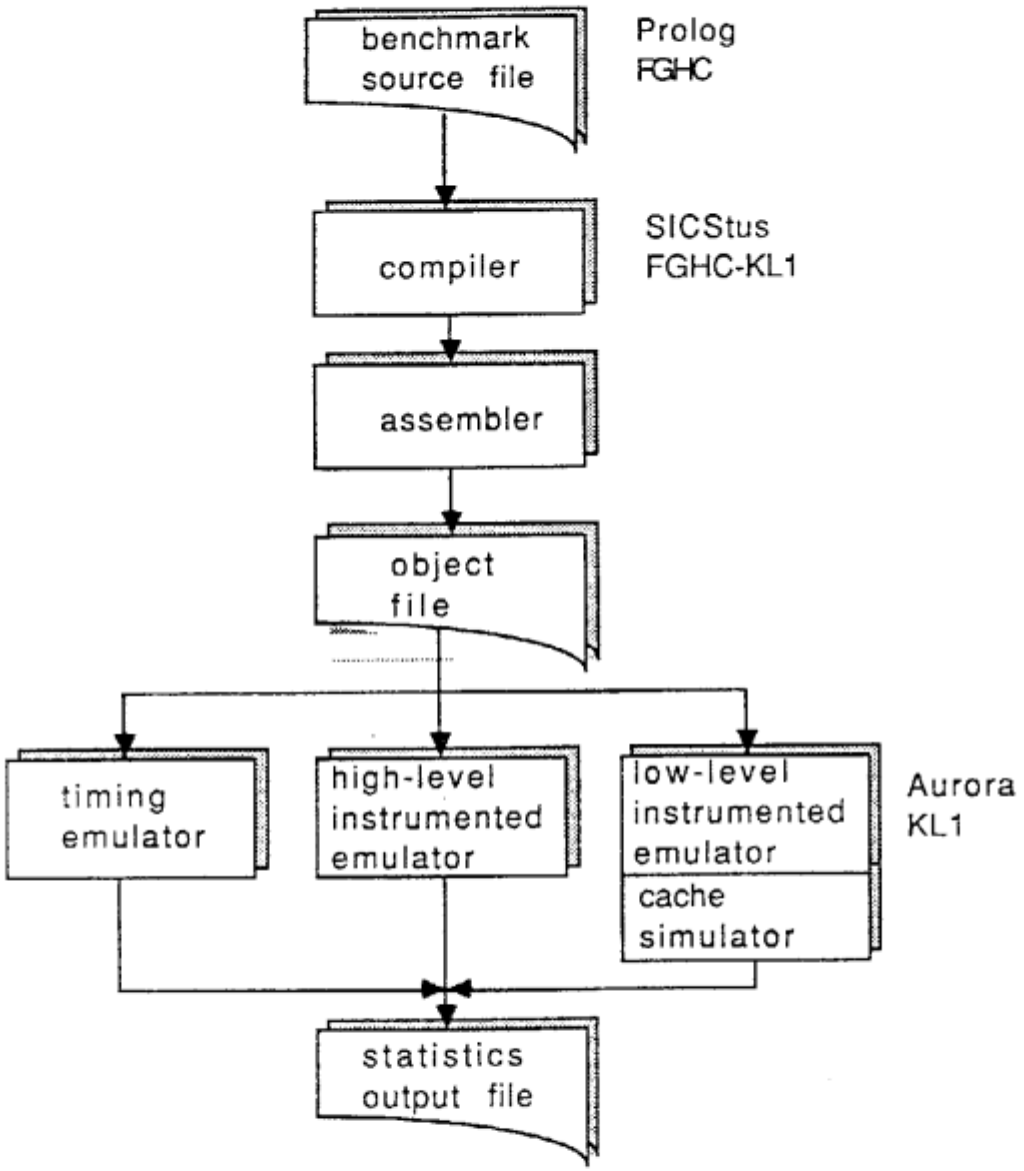


Figure 3: Parallel Logic Programming Architecture Study Methodology

1. **Timing**—measure the raw execution time of the architecture, e.g., to determine speedups.

2. **Instrumented (high-level)**--measure the high-level execution characteristics of the architecture, e.g., number of procedure calls.

3. **Instrumented (low-level)**—measure the low-level memory and cache referencing characteristics, e.g., traffic ratio of a shared-memory multiprocessor model.

These three levels, each a successive refinement of the previous, are now described in more detail.

5.1 Timing

Measuring raw execution time of an emulator for a given architecture on a host machine permits a *gross* comparison of systems performance. In one sense, raw timings are the absolute measure of an architecture. However, the high-level logic programming architectures discussed in this paper are not well-mapped onto current shared memory multiprocessor hosts. For example, the host used in this study—the Sequent Symmetry—uses a write-through cache to ensure cache coherency. Other types of broadcast copyback caches would reduce bus traffic and perform more efficiency. Such a handicap affects different architectures to varying degrees. Other mismatches involve optimizations beneficial to the specialized architectures that are absent from the general-purpose host. For example, Aurora Prolog architecture, based closely on the Warren Abstract Machine (WAM), can benefit greatly from a small set of *shadow registers* for implementing shallow backtracking[56]. Likewise, KL1 can benefit greatly from hardware assisted incremental garbage collection based on the MRB method [12], hardware assisted meta-control, etc. Both architectures can greatly benefit from an increased word size, so that a tag can be included. In addition, KL1 requires a lock bit and possibly MRB within each word.

These differences between the host and the emulated architecture lessen the importance of raw timing measurements. However, for gross comparison the raw timings are valuable. Often timings are used to prove the ability of the architecture to exploit parallelism efficiently. The Holy Grail in this game is "linear speedup," i.e., the ability to execute twice as fast on two PEs, four times as fast on four PEs, etc. Speedup however is a deceptively complex statistic. A common definition, referred to in this study as *relative speedup* is the ratio of the execution time of the program/architecture running on multiple PEs to the execution time of the *same* program/architecture running on a single PE. In this definition, all the overheads of parallel execution remain in the single PE timing, so that good speedups are somewhat easier to achieve.

Another definition of speedup, referred to in this study as *absolute speedup* is the ratio of the execution time of the program/architecture running on multiple PEs to the execution

time of the *fastest sequential program/architecture* running on a single PE. In this case, the single PE measurement does not contain the overhead of parallel management, nor does the algorithm necessarily even support parallelism. Using this definition, good speedups are difficult to achieve.

In this study, measurements are presented for both relative and absolute speedups for Aurora and KL1. For Aurora, the SICStus V0.6 Prolog system is used as a baseline with which to measure absolute speedup. For KL1 no related sequential architecture exists. Therefore an artificial architecture was created, from the parallel KL1 system, wherein most overheads of parallel management were removed. These overheads include locking/unlocking and complex dereferencing.

5.2 High-Level Instrumentation

The Aurora and KL1 systems have been instrumented for high-level statistics by P. Szeredi of Manchester University[51] and M. Sato of ICOT. The instrumentation in both cases consists of software counters inserted throughout the system to collect event tallies. These counters do not greatly disturb parallel execution and therefore present a fairly accurate picture of program characteristics. The dynamic statistics (of interest to this study) collected in these systems are listed below.

- reductions: number of procedure calls executed.
- instructions: number of abstract machine instructions executed.
- backtracks: (Prolog only) number of clause failures causing execution of an alternative clause.
- suspensions: (FGHC only) number of procedure calls forced to suspend (due to synchronization).

5.3 Low-Level Instrumentation

The coherent cache simulator used in this study was written by A. Matsumoto of ICOT. The coherent cache protocol used in this study is documented in [35]. In that study however, a psuedo-parallel cache simulator was used. Here, the simulator has been extended to run in parallel, i.e., when integrated into a system such as Aurora or KL1, there is one cache process (simulating a local cache) for *each* system process (emulating a worker). When running on Symmetry, each worker/cache pair executes on a dedicated host processor. The caches communicate via shared memory. To perform the coherency protocol, caches must synchronize

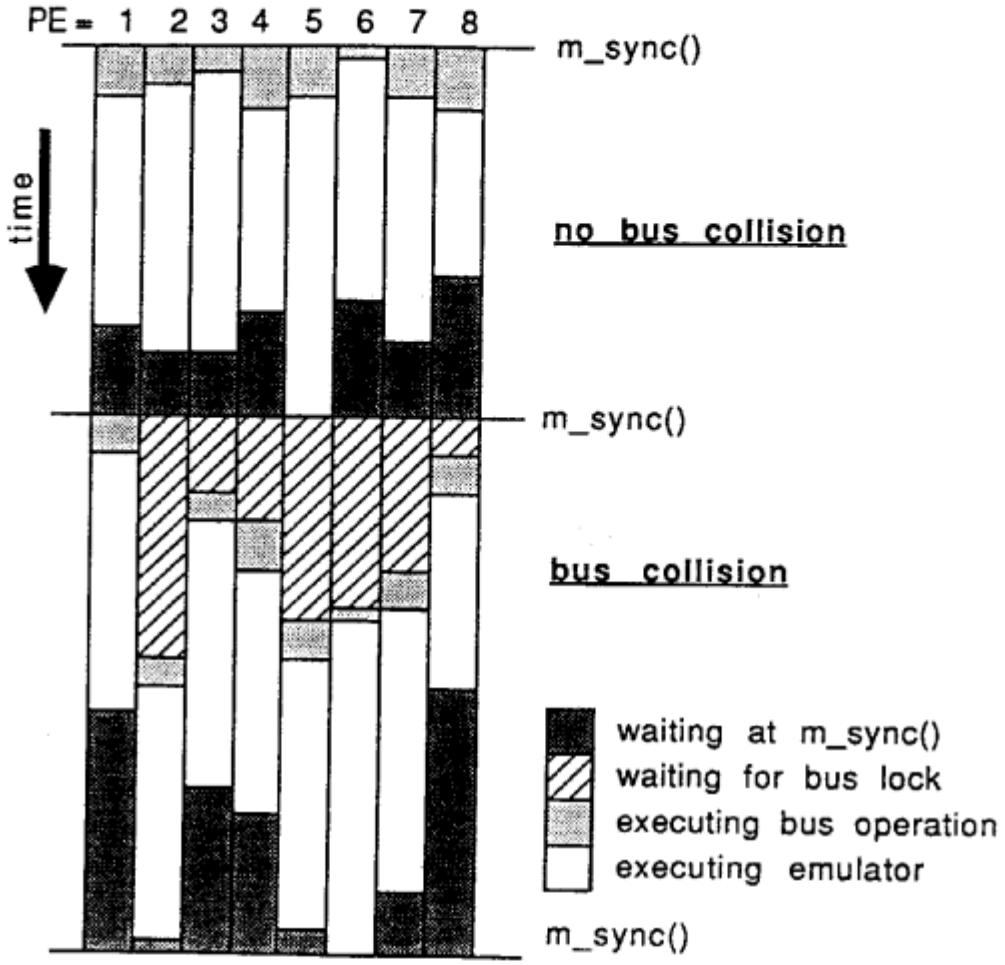


Figure 4: Timing Diagram of Low-level Instrumented System on Eight PEs

when making a simulated bus request. This is to ensure that requests for shared blocks are properly detected by snooping caches. This synchronization is implemented on Symmetry by an `m_sync()` library call (barrier synchronization). The effect of this call is to force all PEs to wait inside the cache simulator, just before the code which simulates a bus request, for all PEs to arrive. When the last PE arrives at this location, they may proceed.

Figure 4 illustrates the execution of the cache simulator interacting with an abstract emulator on eight PEs. Time proceeds vertically. A bar is shown for each PE representing the type of work it is executing: waiting at the `m_sync()` for barrier synchronization, waiting for a bus lock (necessary to avoid races when processing the bus requests), executing the bus request, and executing inside the emulator. The top of the diagram illustrates the most common case when there is no bus collision. A bus collision occurs when two or more PEs make a bus

request for the same address. After each `m_sync()`, each PE checks a common bus request vector and determines independently if a bus collision has occurred. If there is no collision, the cache simulators proceed in parallel as illustrated. If there *is* a collision, all cache simulators attempt to lock the simulated bus. This action sequentializes the bus operations; however, as each completes, the corresponding emulator is reentered and continues executing (see the lower portion of Figure 4). Because bus collisions are very infrequent, the emulation proceeds efficiently.

The barrier synchronization inside the cache simulator has many implications. First, it artificially forces the program to execute in a manner that retains the timing of the non-instrumented system. In other words, one PE is not allowed to execute a series of reductions while other PEs are slowed down due to instrumentation. In previous simulations of KL1, psuedo-parallel simulators where used, wherein a process switch was taken at each reduction. Such simulations retained only a coarse-grain approximation to the original parallelism in the program. In addition, such round-robin task switching disallows the accurate measurement of locking and other time critical events. Ideally the ultimate in accuracy is a system that synchronizes at each simulated machine cycle. This was not implemented because the overhead of such frequent synchronization is excessive. Instead, we chose a synchronization granularity between a reduction and a machine cycle: a bus request. This choice also fits nicely into the requirements of the cache simulator. Even this compromise has a high overhead in terms of simulation time. The fully instrumented systems measured in this study executed at about 50–100 times slower than corresponding non-instrumented systems.

Another important implication of barrier synchronization is the potential is creates for deadlock and livelock. The instrumented systems do locking in order to synchronize parallel processes within their architecture model. For Aurora, this locking is coarse-grain, at the level of locking a node in the OR-tree representing the problem space. The node is usually locked for a significant period of time while a worker process accesses and/or updates the status of the node in an effort to begin executing that branch of the tree. For KL1, locking is fine-grain, at the level of locking a single variable. The variable is usually locked only long enough to check its tag and then possibly bind it.

In both cases, the abstract-level locking and the cache-level barrier synchronization can interfere with each other to cause deadlock and livelock. If an abstract-level lock is set by a PE, which subsequently does a simulated cache reference that causes a simulated bus request, that PE will hang, waiting for synchronization. However, another PE, at the abstract emulation level, may require the abstract-level lock previously set. The second PE will hang waiting for the lock to be freed. Thus deadlock ensues. Livelock can occur when one PE is waiting in the cache simulator when another PE becomes idle and enters the scheduler looking for work to do.

```
typedef struct {
    int     data;
    short   tag;
    char    safe;
    slock_t lock;
    } abstract_word;

#define MY_UNLOCK(x)        {(x)->safe = 0; m_sync();}

#define MY_LOCK(x)                                                    \
                    for(;;){                                          \
                       if ((x)->safe == 1) {                          \
                           m_sync();                                  \
                           continue;                                  \
                           } else {                                   \
                           S_LOCK(&((x)->lock));                      \
                           if ((x)->safe == 0) (x)->safe = 1;         \
                               else {                                 \
                               S_UNLOCK(&((x)->lock));                \
                               continue;                              \
                               }                                      \
                           S_UNLOCK(&((x)->lock));                    \
                           break;                                     \
                           }                                          \
                    }
```

Figure 5: Special Lock Macros for Cache Instrumentation

In general, the schedulers of these systems are quite complex, and make scheduling decisions is based on many factors. It can occur that because one or more PEs are hung at a level lower (the cache simulator) than that understood by the scheduler, a confused decision is made to essentially do nothing. Thus some PEs are hung in the cache simulator and others are looping aimlessly in the scheduler—livelock.

In general, deadlock can be prevented by never placing a call to the cache simulator *inside* a lock interval in the emulator. Livelock can be prevented by carefully placing `m_sync()`s inside the scheduler idle loops. This method of deadlock prevention is not difficult for a system like KL1 with short lock intervals that are easy to spot inside the system code. However, for systems like Aurora, this method is almost impossible to implement.

In both systems, a different deadlock prevention method is used here. A new set of lock/unlock macros has been defined in C that is used to "protect" the lower-level Symmetry lock/unlock library functions. These macros force a PE waiting for an abstract-level lock to issue `m_sync()`s. These `m_sync()`s will kick other PEs out of their calls to the cache simulator, and keep the emulation progressing. The macros used for KL1 are defined in Figure 5 (Aurora macros are similar, but used only for nodes in the OR-tree). The abstract machine word is defined first. It consists of a data and tag field, followed by two lock bytes. The `lock` byte is the official Symmetry lock. The `safe` byte is a soft-lock permitting control over the busy-wait.

5.3.1 Cache Protocol

The cache modeled in this study is a copyback broadcast cache with write allocation (if a write request misses in the cache, the target line is fetched from memory and allocated in the cache). The broadcast protocol, described in detail in Matsumoto [35], involves a five state automata: EM (exclusive modified), EC (exclusive clean), SM (shared modified), S (shared), I (invalid). In addition, a lock directory is assumed, separate from the cache directory. The lock directory is managed in three-states: L (locked), LW (waiting for lock), E (not locked). The cache protocol is based most closely to Bitar's model [2], i.e., modifications to shared data cause *invalidations* to be broadcast to other caches. In addition, when transferring a dirty line from one cache to another, shared memory is *not* updated.

Matsumoto argues why the invalidation protocol is best for the KL1 architecture. KL1 sharing of data is very fine-grained, usually the communication of a logical variable between a single producer process and a single consumer process. Thus broadcast of updated values is not necessary, i.e., shared data is rarely reused over and over. In Aurora, data sharing has vastly different characteristics. The node tree is shared by all processes, who make frequent accesses. Thus a update broadcast protocol seems to be a better choice for Aurora than an invalidation protocol. Unfortunately, the cache simulator available for this study does not implement update broadcast, and so invalidate broadcast was used for all simulations, including Aurora.

The memory operations simulated in this study are a subset of the operations offered by the cache simulator. These are referred to as: `R,W,DW,LR,UW,U`. Read (`R`) and write (`W`) have obvious meanings. For locking, `LR` is lock and read and `UW` is write and unlock. In addition, simple unlock is used (`U`). The KL1 system makes extensive use the optimized `UW` operation. This is necessary because locking is very fine-grained. The Aurora system uses only the standard unlock (`U`) operation. The `UW` optimization is not necessary because locking is coarse-grained.

The final operation used in this study is `DW`, the direct write operation. Direct write is effectively a memory write; however, if the write misses in the cache, the cache will *not* fetch the target line from memory. Instead, the line is allocated in the cache without initialization. Direct write is used when creating new data objects on the top of a memory stack of some sort. Since the architecture knows *a priori* that the memory will be overwritten, fetching of lines from memory can be avoided, and the cache allocated directly. This optimization is used in both the KL1 and Aurora systems.

It should be noted that the cache simulator offers other optimized operations that are not used in this study. These operators were designed specifically for KL1, and are not especially useful for Aurora. Matsumoto claims that all optimizations combined, excluding `DW`, afford only a 6% reduction in bus traffic. Furthermore, his measurements indicate that `DW` alone offers a 31% reduction in bus traffic. These characteristics are specific to KL1, where most bus cycles

BUS-WIDTH	1	1	1	1	1	2	2	2	2	2
MEM-ACC-TIME	8	7	6	5	0	8	7	6	5	0
FROM-GM-SOUT	13	12	11	10	10	11	10	9	8	6
FROM-GM-ONLY	13	12	11	10	5	11	10	9	8	3
MCTOC-SOUT	10	10	10	10	10	6	6	6	6	6
MCTOC-ONLY	7	7	7	7	7	5	5	5	5	5
CCTOC-SOUT	10	10	10	10	10	6	6	6	6	6
CCTOC-ONLY	7	7	7	7	7	5	5	5	5	5
SOUT-ONLY	5	5	5	5	5	3	3	3	3	3
INV-ONLY	2	2	2	2	2	2	2	2	2	2

Table 1: Shared Memory Multiprocessor Bus Models (units in bus cycles)

are due to heap referencing, a result of a free-list style of memory management. Characteristics of Aurora are different – the major contributer to bus traffic is the code area and the control stack. Because Aurora memory management is based on stacks (e.g., free-lists in KL1), the DW operation can be used without the help of other special operations to reduce the memory bandwidth requirement.

5.3.2 Shared Memory Models

The cache simulator utilizes an internal model of a shared memory multiprocessor to calculate the bus traffic generated by the benchmark program. For each of various bus operations, a certain number of bus cycles is required. The requirement is based on the assumed bus width, the main memory access time, and the sophistication of the bus manager. In Table 1, ten alternative models are presented, differing in these parameters. At the top of the table, bus widths of one and two words are listed, as well as main memory access times of zero and 5-8 cycles. The 5–8 memory access time models use a simple bus model wherein bus operations cannot be overlapped in any way. The zero memory access time models are special in that they estimate the performance of a sophisticated bus that can overlap operations (thus the effect of waiting to access memory disappears). Listed in Table 1 are the number of cycles required for each bus action. FROM-GM-SOUT is fetching a block from global memory while swapping out a block from cache. FROM-GM-ONLY is fetching a block from global memory only without swap-out. Similarly, MCTOC is a cache-to-cache transfer of a modified block. CCTOT is a cache-to-cache transfer of a clean block. SOUT-ONLY is a swap-out of a block from cache to global memory. INV-ONLY is an invalidation of one cache by another.

A major point to note is that as memory access time is decreased, the bus operation cycles do not decrease proportionally. This is because only FROM-GM operations access the

memory, and even those operations have overheads that overshadow the access time. As will be seen in the measurements of KL1 where cache-to-cache transfers are heavy, faster global memories do not significantly decrease bus traffic. In contrast, by increasing bus width, the bus operation cycle times in Table 1 decrease significantly. This observation is also supported by measurements presented in later sections.

5.3.3 Sample Cache Simulator Output

A discussion is now given explaining the output of the cache simulator. For this discussion, a sample of the output is presented, broken down into its constituent parts and annotated. This is the output of a test program running on the Aurora OR-Parallel Prolog emulator. The header below gives this information, including additional cache parameters: eight processing elements (PEs), 64 columns, four sets per column, four words per block (line), one sub-blocks per block. In the simulations analyzed in this paper, only number of columns was varied from this organization.

```
Aurora OR-parallel Prolog
8 PE -- c64,s4,w4,t1

GVNPTCL 00000001, GVNGMOD 00000003, BYTOWD  00000002, AURORAM 00000001
GVNPE   00000008, GVNSET  00000004, GVNCOL  00000040, GVNBLK  00000004
GVNSECT 00000001, BUSWIDT 00000001, MACCTIM 00000008, CTCXTIM 00000001
INVTIME 00000002, GOALRPS 000000c8, GOALCYC 00000032
```

The next table shows the breakdown of memory references to different areas in the abstract machine. Note that for Aurora, UW is not used.

```
TABLE GIVEN-CPU-COMMAND(AREA)
GVNCMD      HEAP     INST      ENV     NODE      LBA      GBA     TRAIL     TOTAL
R        1908656  2184800   163550  1461802    33103   601789   1138577   7492277
W          13010        0    12746   910418    30806   586681   1127825   2681486
DW        153204        0        0        0        0        0         0    153204
LR             0        0        0     3545        0        0         0      3545
UW             0        0        0        0        0        0         0         0
U              0        0        0     3545        0        0         0      3545
TOTAL    2074870  2184800   176296  2379310    63909  1188470   2266402  10334057
```

The next table shows the breakdown of bus operations to different areas in the abstract machine. The three bus operations are: fetch (F), fetch and invalidate (FI), and invalidate (IV). Fetch is used for instance on a read (R) miss. Fetch and invalidate is used for lock and read (LR) operations and write (W) misses. Invalidate is used for a write (W) or lock and read (LR) cache hit on shared data.

In the data below, we see that trail referencing has excellent spatial locality, missing less than 1% in the cache. In contrast, instructions appear to have the least locality, missing over 5% of its references. We also see that references to nodes favor write misses whereas references

to the heap and to environments favor read misses. Note that the control stack (nodes) exhibits a high invalidation (IV) count, indicating (as one would expect) that sharing of the OR-tree is common among the PEs. In contrast, the heap and environment stacks show almost no sharing.

TABLE ISSUED-BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
F	63630	121159	5543	11392	1055	7978	11196	221953
FI	183	0	29	17709	296	6826	8738	33781
IV	33	0	5	1490	0	0	267	1795
TOTAL	63846	121159	5577	30591	1351	14804	20201	257529

The previous three bus commands are decomposed into bus operations dependent on the state of the data (if in the cache). The following table gives the breakdown of the bus operations across the abstract memory areas. Thus we see for instance that cache-to-cache copy without swap-out (CCTOC-ONLY) for instruction references is the most frequent operation. Some of the operations have no counts because their corresponding cache commands are not used (these are the optimizations mentioned earlier). For this program, we see that the instructions generate the most bus operations.

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	844	1786	452	3541	149	2546	3293	12611
13:FROM-GM-ONLY	4034	5018	4958	18933	1202	12258	15552	61955
10:MCTOC-SOUT	96	0	5	139	0	0	34	274
07:MCTOC-ONLY	395	0	61	2746	0	0	284	3486
10:CCTOC-SOUT	14234	30950	13	139	0	0	122	45458
07:CCTOC-ONLY	44210	83405	83	3603	0	0	649	131950
05:SOUT-ONLY	814	0	0	0	0	0	0	814
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	33	0	5	1490	0	0	267	1795
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	64660	121159	5577	30591	1351	14804	20201	258343

The following table multiplies the previous bus operation counts by cycle times. The cycles times, listed at the left-hand-side of the table, are derived from a simple model of shared memory. The model shown below assumes an eight cycle shared memory access time and a one word wide bus. Although this specific model may not be the most realistic, this table gives insight into the trouble spots of the architecture. For example, instructions and heap are far and away the biggest burners of bus bandwidth. In addition, the cache-to cache transfers are by far the most frequent operations.

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	10972	23218	5876	46033	1937	33098	42809	163943
13:FROM-GM-ONLY	52442	65234	64454	246129	15626	159354	202176	805415
10:MCTOC-SOUT	960	0	50	1390	0	0	340	2740
07:MCTOC-ONLY	2765	0	427	19222	0	0	1988	24402
10:CCTOC-SOUT	142340	309500	130	1390	0	0	1220	454580
07:CCTOC-ONLY	309470	583835	581	25221	0	0	4543	923650

05:SOUT-ONLY	4070	0	0	0	0	0	0	4070
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	66	0	10	2980	0	0	534	3590
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	523085	981787	71528	342365	17563	192452	253610	2382390

The following table displays the total number of cache operations (right hand column), broken down into hits and misses. The first four columns of the table further break down the cache hits by the state of the hit data: exclusive (clean EC, and modified EM) and shared (clean SC, and modified SM). Note that unlocks (U) rarely miss, as we expect. Note that DW rarely misses, i.e., the line is already allocated in the cache. Therefore direct write has little beneficial effect in this program.

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	233619	3484591	3538973	13141	7270324	221953	7492277
W	23618	2739897	638	497	2764650	31733	2796383
DW	164	34094	4	13	34275	4015	38290
LR	164	673	510	150	1497	2048	3545
UW	0	0	0	0	0	0	0
U	310	2380	1	853	3544	1	3545
TOTAL	257875	6261635	3540126	14654	10074290	259750	10334040

The following table breaks down all cache misses (right-hand column) into where the data is retrieved from. Recall the protocol is a write allocate policy. The areas of retrieval are from another cache (FRCACHE) and from shared memory (FROM-GM). Cache retrieval is further broken down (first two columns) into FRMC (from modified cache) and FRCC (from clean cache).

We observe for this benchmark that most missed lines are retrieved from another cache instead of shared memory, by a ratio of over 2:1. In addition, almost all cache-to-cache transfers are clean. Therefore organizations should concentrate on making clean cache-to cache transfers fast, possibly at the expense of memory-to-cache transfers.

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	3760	175467	179227	42726	221953
ER	0	0	0	0	0
RP	0	0	0	0	0
W	0	57	57	31676	31733
DW	0	0	0	4015	4015
LR	0	1884	1884	164	2048
UW	0	0	0	0	0
U	0	0	0	1	1
TOTAL	3760	177408	181168	78582	259750

The following table gives a summary of the effectiveness of the DW operation. The GIVEN count is the number of direct writes requested by the PEs. The ISSUED count is the number of DW operations with a line address, i.e., an address that is a multiple of a cache line. The ISSUED

count is the actual number of cache line allocations saved by the DW optimization. Other DW requests are simply treated as normal write (W) requests. The final two rows in this table give the number of ISSUED DW operations that did (not) require swap-out. These statistics show that direct write is saving about 4000 line transfers from shared memory. Given the total number of misses (about 260,000), direct write reduces memory bandwidth requirement very little.

```
TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA
GIVEN                153204
ISSUED                38291
WITHOUT-SWAP-OUT       3201
WITH-SWAP-OUT           814
```

The following table calculates the miss ratio of the simulation. The definition of miss ratio is somewhat complicated by the direct write operation. The definition described here is due to Matsumoto. Direct writes that miss in the cache do not require fetching the target line from memory. The allocation of the target line may require however the swap-out of a resident line. For the calculation of miss ratio, direct write misses without swap-out (DW-WITHOUT-SOUT) are considered as hits. Other direct write misses (with swap-out) are considered as misses. This is a somewhat conservative definition of miss ratio. The second row gives the total number of hits and the third row gives the total number of misses. Hit and miss ratios are then calculated by dividing these totals by the total number of memory references. We observe again that for this program, direct write has almost no effect on reducing miss ratio.

```
TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA
          3201         DW-WITHOUT-SOUT
      10077491         T-HIT + DW-WITHOUT-SOUT
        256549         T-MISS - DW-WITHOUT-SOUT
         97.52 [%] HIT-RATIO
          2.48 [%] MISS-RATIO
```

The following three tables give low-level characteristics of cache operation. The first shows a snapshot of the cache directory (what states) at the end of program execution. The second shows a snapshot of the cache directory (what memory areas) at the end of program execution. The third table shows the occurrence of bus collisions in the cache simulator. Recall that the cache simulator synchronizes PEs before every simulated bus operation. A bus collision is a simulation cycle wherein two or more PEs perform an operation on the *same* address. To cope with this situation, some of the PE requested operations must be transformed to retain consistency. For example, if two PEs both issue a write request to the same address, the PE that is serviced first by the simulator will issue an IV (invalidate) bus command, invalidating the other PE. The second PE also issued an IV command, but this is no longer correct. The IV command is transformed by the simulator into an FI (fetch and invalidate) command. We observe no collisions in this benchmark, indicative of little sharing.

```
TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution
      EC       EM       SC       SM        C        1  UNUSED    TOTAL
     190      455     1202       72        0      129       0     2048

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution
    HEAP     INST      ENV     NODE      LBA      GBA    TRAIL INVALID    TOTAL
     427      417       30      468       29      180      368      129     2048

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION
     HEAP      INST       ENV      NODE       LBA       GBA     TRAIL      TOTAL
        0         0         0         0         0         0         0          0
```

The following portion of the cache output is devoted to the *bus traffic ratio* (BTR). BTR is defined as the total number of bus cycles (BC) divided by the total number of memory references (MR):

$$BTR = BC/MR$$

Thus BTR has the units cycles/reference and is not rigorously a "traffic ratio". This definition is useful to compare different systems. Unfortunately, in itself, BTR does not indicate the reduction in bandwidth requirement afforded by a cache. The reader may wish to use the following statistic:

$$BTR' = \frac{BC}{T_a \cdot MR}$$

where T_a is the memory access time. Alternatively, the reader may wish to use Matsumoto's *bus usage ratio* (BUR) statistic [35]:

$$BUR = \frac{BC \cdot T_{bus}}{I/P}$$

where I is the total number of executed instructions and P is the gross execution rate in units of instructions per second. These three statistics give differing views of the same thing. In this study, we use BTR only because our main objective is to compare two systems. We informally refer to the BTR as a "traffic ratio."

Below, several BTRs are calculated for alternative organization models. The results for this program show that bus width is the most important factor in determining traffic ratio. Memory access time is rather unimportant because as shown in earlier statistics, most traffic is cache-to-cache. This shows that relatively slow shared memories can be used without detrimental effect, but that high bus bandwidth is required. In addition, the sophisticated (overlapped operation) bus model (MEM-ACC-TIME = 0) offers a significant reduction of traffic with respect to the non-overlapped model. This gap is most significant for wider bus models. All the benchmarks studied follow these trends.

```
TABLE BUS-TRAFFIC-RATIO
BUS-WIDTH[W] MEM-ACC-TIME    MEM-REF BUS-CYCLE TRAFFIC-RATIO
           1            8   10334057   2382390         0.231
           1            7   10334057   2307824         0.223
```

c	size(dir)	size(data)	size(total)
32	3648	20K	24128
64	7040	40K	48000
128	13568	80K	95488
256	26112	160K	189952
512	50176	320K	377856

Table 2: Cache Sizes Simulated (in bits)

```
1        6   10334057   2233258        0.216
1        5   10334057   2158692        0.209
1        0   10334057   1848917        0.179
2        8   10334057   1777830        0.172
2        7   10334057   1703264        0.165
2        6   10334057   1628698        0.158
2        5   10334057   1554132        0.150
2        0   10334057   1219135        0.118
```

The cache sizes measured in this study range from 512 words to 8K words. The abstract architecture word size need not be specified. It suffices to assume that both the Aurora and KL1 architectures have approximately the same size word. Abstract machine logical addresses are assumed to be four bytes. For the purpose of presenting cache statistics as functions of cache size, the abstract machine word is assumed to be 5 bytes. Cache size is calculated as the sum of the cache directory size, cache data area size, status bits and least-recently-used (LRU) bits. The model of cache size shown below is due to A. Goto[24]. For all simulations, we assume 40 bits per word ($b = 40$), four words per block ($w = 4$), four blocks per set ($s = 4$), one sub-block per block, three status bits per block ($n = 3$), and two LRU bits per column ($l = 2$). The number of columns (c) is varied. Table 2 shows the cache sizes measured in this study.

$$
\begin{aligned}
x &= ((32 - log(w) - log(c)) + n) \times s + l \\
y &= c \\
size(data) &= x \cdot y \\
size(dir) &= b \cdot c \cdot w \cdot s \\
size(total) &= size(data) + size(dir) = xy + bcws.
\end{aligned}
$$

5.4 Benchmarks

The benchmarks studied here are parallel solutions of small symbolic manipulation problems. This is by no means a complete cross-section of the types of problems that can be solved with

logic programming systems; however, the problems do represent a subset of these applications. The benchmarks were necessarily kept small to facilitate their construction, debugging and execution on the unstable systems used. In addition, program solutions in both languages were required for each problem, and therefore small problems were chosen.

Descriptions of the benchmarks are listed below. See the Appendix for a complete source listing of each program.

- **Triangle**—finds all (133) winning solutions to a triangular peg game. This problem is the same as R. Gabriel's Lisp benchmark [23], except that here, three initial moves are taken on the triangular board. This initialization is necessary to reduce the problem space to a reasonable size. See Tick [54] for the sequential Prolog version of this program. The FGHC program was translated from the Prolog by an automatic "continuation" based method as described by Ueda [59]. Post translation optimizations were then performed by hand [22].

- **Puzzle**—finds all (65) solutions to a puzzle packing problem. This problem is based on R. Gabriel's Lisp benchmark, but modified to drastically reduce the problem space, allowing search for all solutions. Here we pack a 5x4x3 solid (with corner missing) with seven pieces: (3) 3x2x1, (2) 1x3x1, (1) 3x3x1, (1) 4x2x1. See Tick [54] for the sequential Prolog version of the original form of this problem. The FGHC version was written by the author.

- **Pascal** generates the 100th row of Pascal's Triangle, using integer bignums to represent the coefficients. The maximum coefficient of the 100th row is represented as `[97256, 48124, 19333, 45564, 13445, 10089]`. The Prolog version of this program was written by the author, using the bignum library from DEC 10 Prolog, written by R. O'Keefe of Quintus Computer Inc. The Prolog program uses a optimized form of M. Carlsson's hack for implementing AND-in-OR parallelism [10]. See Takagi [52] for the original version of the FGHC program, written by E. Sugino. The version measured here includes an FGHC translation of the (integer addition) bignum library.

- **Semigroup** generates all (813) members of a Bradt Semigroup B2, given a set of four generators. The elements of the semigroup are lists of length 10. The Prolog version of this program is a modified version of the original written by R. Overbeek of ANL [19]. The FGHC version was written by N. Ichiyoshi of ICOT.

- **Queens** finds all (724) solutions to the 10-Queens problem. There are several queens algorithms measured, distinguished by the initials of the names of their authors. **HKqueen**, written by H. Kondo of NTT, uses constraints implemented via logical variables in Prolog.

IBqueen, written by I. Bratko [5], uses constraints implemented with lists of diagonal offsets (in Prolog). **MBqueen**, written by M. Bruynooghe, uses a fused generate and test algorithm (in Prolog—this is the "classic" Queens program). **KKqueen**, written by K. Kumon of Fujitsu, is a stream-based FGHC program. **AOqueen**, written by A. Okumura of ICOT, is a layered-stream-based FGHC program [39]. **KUqueen**, developed by K. Ueda of ICOT, is a continuation-based translation of **MBqueen** [59].

Note that these benchmarks are not "as is" programs taken from an authentic user base. Instead the benchmarks were carefully rewritten to perform efficiently. A structure-based version of **Triangle** was determined to be inferior to the list-based version given here. The KL1 version of **Triangle** was hand-optimized after its translation from Prolog. A list-based version of **Puzzle**, due to L. Sterling, was determined to be inferior to the structure-based version given here. The KL1 version was rewritten countless times to increase its efficiency. Initially, a Prolog version of **Pascal** written by Sugino was used, but this did not permit exploitation of OR-parallelism. The refined version presented here uses unrolling to increase the coarseness of AND-parallel goals to allow AND-in-OR parallel techniques. **Semigroup** was optimized several times by Overbeek and Ichiyoshi, in both languages.

Several versions of **Queens** were also developed—in this case, some of the programs are included here to give further insight into how changes in algorithm can radically effect the interpretation of cache simulation results. **HKqueen** and **AOqueen** are most often compared because they represent the fastest algorithms in Prolog and FGHC respectively. The lesson taught by this simple benchmark is that parallel algorithms for the same application vary greatly in performance (more so than sequential algorithms), and thus analysis of a large set of algorithms for a set of given applications is necessary to fully understand parallel architecture tradeoffs.[9]

The various solutions to these problems, in both Prolog and FGHC, are summarized in Table 3. The table gives the number of static procedures, source lines, and clauses. Dynamic measures are given for the execution time on eight PEs (in seconds) on a Sequent Symmetry, the speed-up on eight PEs (relative to the same program on a single PE), and the number of procedure *entries* executed. For Prolog, a procedure entry is either a reduction (procedure call) or backtrack. For FGHC, a procedure entry is either a reduction or suspension. As can be seen, the programs are small—this is a major limitation of this study. Although the amount of computation of the programs is sufficient to exercise the cache simulators, the benchmarks do not have large working sets as do big applications like CAD, natural language, compilers, etc.

[9] **IBqueen** is not included in many measurements presented here because it takes too long to execute, given its serious inefficiencies. Whereas **HKqueen** required 11,623,125 data references to execute 10-queens, **IBqueen** required 92,296,280.

benchmark	procs	lines	clauses	seconds	speedup	entries
Prolog						
Triangle	5	86	43	12.0	7.7	587037
Semigroup	21	126	47	18.6	3.2	153555
Puzzle	13	233	33	6.3	7.6	145106
Pascal	42	286	74	41.7	2.0	267276
HKqueen	7	37	12	14.3	5.6	334782
MBqueen	8	16	15	21.5	6.6	772779
IBqueen	8	27	14	82.1	7.3	4996096
FGHC						
Triangle	45	182	87	49.3	5.8	320114
Semigroup	12	104	63	87.5	4.8	292307
Puzzle	13	151	51	55.3	6.5	852608
Pascal	51	310	153	16.6	6.1	320113
AOqueen	7	43	22	27.3	6.8	361894
KKqueen	6	26	15	11.5	7.3	873419
KUqueen	9	34	19	45.6	7.3	1026142

Table 3: Short Summary of Benchmarks

The programs all have significant parallelism and most can exploit that parallelism efficiently (with the exception of **Semigroup** and **Pascal** in Prolog). In general, the OR-parallel Prolog programs display less parallelism than the FGHC programs. Looking at procedure entries and raw execution time, in general, the Prolog programs do less work than the corresponding FGHC programs.

When FGHC performs more procedure entries it is characteristic of the lower semantic power of the language as compared to Prolog. Prolog can exploit full unification coupled with backtracking to solve many of these problems quite efficiently. FGHC is limited to one-way unification and must "emulate" backtracking at the source language level. Note that although **Triangle** performs more Prolog procedure entries, the Prolog executes about four times faster than FGHC. In **Puzzle**, the difference is more pronounced. In both programs, Prolog can use unification of logical variables to avoid the structure copying necessary in FGHC. Comparing the two fastest Queens algorithms, **HKqueen** and **AOqueen**, we find that Prolog's ability to backtrack over unification gives it a 2:1 speed advantage, whereas procedure entries are almost equal.

The remaining two benchmarks, **Semigroup** and **Pascal**, solve single-solution problems. Prolog **Semigroup** uses a 2-3 tree [5] to store the elements of the semigroup. This sequentializes the search for an element, but the search is quite efficient. KL1 uses a pipeline of filters to store the elements of the semigroup. This parallelizes the search for an element (different searches can be pipelined), at the cost of an inefficient (linear) search for each element. **Pascal** has no OR-parallelism, so that Aurora must simulate AND-parallelism via meta-interpretation (this is called AND-in-OR parallelism [10]), at great overhead. The overhead of exploiting AND-in-OR parallelism (FGHC is over twice as fast as Prolog on eight PEs) comes from the bookkeeping needed to execute many fine-grained processes. It should be noted that Carlsson et. al. [10] measured a *maximum* speedup of 2.2 for an AND-in-OR parallel compiler running on Aurora. Although the compiler had coarse-grain parallelism, 30% of the computation was sequential, thus limiting speedup. In general, FGHC can manage fine-grained processes much more efficiently than can Prolog, whereas Prolog can manage coarse-grained processes more efficiently than FGHC. The amount of such parallelism in real applications is a yet unanswered question.

Interesting results of this study are the comparison of the algorithms forced upon the programmer by the language definition. If FGHC encourages object-oriented programming style by the nature of stream communication, then the performance object-oriented programs is important to study. Likewise the various uses of logical variables and backtracking that Prolog encourages are important to measure.

The algorithms chosen are sometimes different in each language, often in definition of data

structures. In **Semigroup**, Prolog's use of 2-3 trees gives it a definite advantage over KL1. The KL1 pipeline process structure can conceivably be rewritten into a tree structure that will speedup up the search. In **Puzzle**, Prolog's use of logic variables obviates the need for copying large data structures, as is necessary in KL1. Because **Puzzle** is an all-solutions search, KL1 *destructive arrays* cannot be used to represent the data structures. This is also true for **Triangle**. **IBqueen** in Prolog is naturally suited for arrays, but arrays are not implemented in Aurora, so inefficient lists are used instead.

Most of the benchmarks chosen in this study perform an "all-solutions search". This means that the problem space contains multiple, independent solutions that must all be found by the program. **Pascal** is a completely determinate program, finding a single solution (calculating a row in Pascal's triangle). Given a problem space containing multiple solutions, all-solutions search is used in a benchmark to avoid unreliable measurements. If for instance only one solution is required from a multiple solution space, a *different* solution may be found when the *same* program is run on different numbers of PEs. This may result in either sublinear or superlinear speedups. This problem is one of determinacy – a good benchmark is a determinate benchmark. Nondeterminacy can cause high variance in performance measurements that is not attributable to the architecture or system, but rather to luck.

Unfortunately, choosing all-solutions search problems give OR-parallel Prolog an advantage over FGHC. Prolog can collect all solutions with builtin functions (such as `findall` and `bagof`) that backtrack over solutions more efficiently than can be simulated in FGHC. An all-solutions search problem guarantee OR-parallel Prolog a source of easily exploitable parallelism. However, OR-parallelism in single-solution problems is not so easily uncovered by Prolog. Two single-solution problems, **Semigroup** and **Pascal**, were chosen to illustrate this contrast. As seen in Table 3, the Prolog solutions to these problems have the lowest speedup of all the benchmarks. It is shown in later sections that the overheads of exploiting even that small amount of parallelism is great in terms of absolute speed and program readability (declarativity).

6 Architecture Models

Instrumentation and analysis of an emulated architecture is an empirical and inexact science. In addition to the mechanical problems (Section 5.3) of simulating the precise timing of the anticipated target host, there is additionally the imprecise nature of the instrumentation. These errors include lack of complete instrumentation, inaccurate order of instrumentation, mismatches between the emulator and target host with respect to storage models, instruction formats, and system overheads (e.g., garbage collection). The inaccuracies present in the systems studied here are outlined in this section, with a discussion about their relative importance.

6.1 KL1

6.1.1 State Space

Modeling a real architecture on a target host, with an emulated architecture on a partially mapped host, requires creating a correspondence between emulator variables and target machine registers and memory. For example, in his study of the Prolog WAM, Tick [55] assumes that the WAM state registers and argument registers are implemented in hardware (on the target host), even though they are actually implemented as C variables in the emulator. An extended model assumes the top choice point of the local stack is also stored in a register file (similar to the Pegasus microprocessor [46]). Such correspondences allow the system designer to evaluate the effect that buffering hardware has on reducing the bandwidth requirement.

In the KL1 measurements presented here, we assume a very liberal correspondence of architecture state to registers. The reason for this is two-fold. Firstly, the emulator code is complex, not documented, written by a different person than the instrumentor, and lacks data abstraction. Thus the ability to determine a minimum necessary architecture state space was difficult. Secondly, even when certain particulars of the emulator were understood, it was often not clear if they were to be considered a fundamentally necessary part of the architecture (for example, see the next section about meta-counts).

For these reasons, most emulator variables were considered either not necessary for the target architecture, or able to be allocated to temporary registers. In addition, the KL1 state variables, as defined by Kimura [30] and defined internally to the emulator, and goal arguments were also mapped onto registers. Note that the number of KL1 state variables defined internally is much larger than described by Kimura, comprising all goal queue pointers, processor status, communication buffer pointers, interrupt status, suspension stack pointers, meta-counts, garbage collection pointers, etc. Assuming these can all be placed in registers is a best case assumption for KL1. Of course, memory references to the major storage areas (heap, goal, instruction, suspension, and communication) were instrumented as target architecture memory references.

6.1.2 Meta-Control

Recall (from Section 2.2) that execution of an FGHC procedure can result in one of three states: success, failure, and suspension. *Meta-control* is a generic name for architecture extensions allowing stronger control of one process (the caller) over another (the callee). For example, an operating system needs to call a user program and have the program status returned: success, failure or deadlock. One such mechanism for FGHC is described by Ichiyoshi [29]. Essentially the difficulty in determining deadlock, or even termination, is that suspended goals "float"

around the storage space, hooked only to the variables they were suspended on. There must be a synchronous method of determining when all the goal queues are empty and if there are no floating suspended goals.

The details of the actual proposed schemes do not concern us here; however, the overheads of these systems do. The benchmarks measured in this study are all single programs with no meta-control. In the KL1 emulator used here, a single program is executed and returns its status to the emulator. A meta-count is a special counter (one per PE) in the emulator used to keep track of called processes. It is from the meta-counts that the program's completion status can be determined. Each reduction, the meta-counts are manipulated to keep track of things. Matsumoto measured this overhead (for single program execution, assuming a given meta-count scheme), and reported that 4% memory references and 15% bus cycles are devoted to this type of bookkeeping.

It is felt that 15% is far too large a penalty for a real system running a single, correct user program with no meta-control. In a real system, compiler optimization and hardware assist would reduce this overhead. In this study, no assumptions are made about meta-control complexity and overheads. We assume that the meta-counts are implemented with hardware registers, and further that simple, single program execution (like the benchmarks discussed here) requires no meta-control memory references. This benefits KL1 as a best case assumption.

6.1.3 Unification and Suspension Stacks

The KL1 architecture uses two small runtime stacks for managing recursive (general) unification and the suspension mechanism. The former is similar to the Prolog unification stack, called the PDL (push-down-list) by Warren [61]. In the case of KL1, each PE has its own PDL, used for (general) active unification, passive unification, and anti-unification. Each PE also has its own suspension stack, used for temporarily storing variables that require output bindings during head unification. References to these stacks are *not* instrumented as abstract machine memory references in this study. The unification stack is expected to display characteristics similar to Prolog, where less than 2% of all data references were to the PDL [56], and spatial locality is extremely high. The suspension stack is expected to also display high spatial locality and small maximum growth.

6.1.4 Spatial Locality

The KL1 and Aurora emulators were instrumented assuming that the abstract machine shared-memory addresses are the same as the Sequent shared-memory addresses. In other words, when

issuing an abstract machine memory request, the Sequent virtual address is issued.[10] This method avoids the necessity to translate each Symmetry address into a KL1 machine address, thus saving simulation time. The method has the disadvantage that locality is somewhat lessened (from what it would be on the target host). There are several places in the emulator where locality is lessened:

- instruction size: the emulator uses large indexing instructions consisting of many words. These instructions waste space, thus the instrumented architecture sees a code space with lower locality than an architecture with an optimized set of formats. This effect is minor.
- goals size: the emulator allocates fixed-size areas for goal records, even though goals have differing numbers of arguments. This causes the memory to be allocated more rapidly than on the target host. However, the unused portion of the goal records are never referenced, and thus cache performance is not significantly affected.
- heap overloading: the emulator allocates goal records and suspension records on the heap, instead of allocating them on independent goal and suspension areas. Thus spatial locality is somewhat lessened. The effect is minor however because these records are multiples of the cache block size and are always allocated on a block boundary. The normal heap data may be interrupted by the inclusion of this extraneous data.

6.1.5 Timing

The lock (LR) operation is issued *after* the Symmetry lock is obtained instead of before. This ensures that the abstract machine lock state does not change prematurely, i.e., before the corresponding Symmetry lock is captured. This increases the accuracy of the simulations.

6.1.6 Direct Write to Goal and Communication Areas

The read buffer and read purge operations of the KL1-specialized coherent cache model[35] were *not* used in this study. Therefore direct writes could not be instrumented to the goal and communication areas. It is anticipated that these optimizations will help reduce the required KL1 bus bandwidth by a significant amount. Because these operations are specific to KL1 and cannot be used in Aurora, it is felt that including them would complicate the comparison, and lessen the fairness, of the cache statistics.

[10] The KL1 emulator manipulates 8 byte abstract words, the expected word size of the ICOT designed PIM [25]. The Aurora emulator manipulates 4 byte abstract words. The Symmetry is byte addressable, so in order to make a fair comparison, the cache simulator shifts KL1 addresses by 3 bits and Aurora addresses by 2 bits. This means that both systems are simulated as if the basic word size of their architectures were equal (whatever size that may be). The cache statistics presented here assume that size is 40 bits.

6.2 Aurora

6.2.1 State Space

Many of the comments in Section 6.1.1 concerning mapping the KL1 state space apply to Aurora also. The Aurora system however has fairly good data abstraction, facilitating this mapping. Still, neither Aurora nor KL1 was implemented with instrumentation in mind, and as a result, hidden pieces of the state space have remained hidden. The Aurora system is split into two major pieces: the worker (essentially a WAM engine) and the scheduler (in this case, the Argonne version). The worker is modeled as is a sequential WAM, assuming that the WAM stack-group state (e.g., B, E, etc.) are implemented in registers. In addition, all temporary variables used in the worker functions, and the complex data structure *defining* a worker are also (liberally) assumed to be mapped onto registers. Assuming these can all be placed in registers is a best case assumption for Aurora. Or course, memory references to the major storage areas (heap, control stack, local and global stacks, trail and binding arrays) were instrumented as target architecture memory references.

6.2.2 Warm Start

The Aurora system is a complete Prolog system that is bootstrapped with a top-level read-eval loop written in Prolog, running on all PEs. This is in stark contrast with the KL1 system where the top-level read-eval loop is implemented inside the emulator, in C, running on one PE before slave PEs are forked. Thus when starting the Aurora system, the system boots itself, requiring the execution of several hundred lines of Prolog code. After the boot, the benchmark object image is loaded and then the benchmark is executed. The entire startup generates about 200,000 memory references, distributed in an unknown fashion across the PEs. This is most significant for **Puzzle**, where it represents 2% of all memory requests. A facility to re-initialize the cache simulator from Prolog was not implemented, and so the Aurora measurements presented in this paper include the effects of this ("warm") start. KL1 measurements are a pure "cold" start. This difference is minor.

6.2.3 Argonne Scheduler Sleep Time

After instrumenting the scheduler, we noticed a drastic, unbelievable increase in control stack (NODE) reads on multiple PEs with respect to single PEs, for the **Semigroup** and **Pascal** benchmarks (where all other reference areas counts remained the same). These benchmarks cannot exploit OR-parallelism efficiently and therefore on multiple PEs, the workers are spending a great deal of time in the scheduler, looking for work.

The Aurora system instrumented for this study uses the Argonne scheduler[7]. The scheduler

has a main loop in which an idle worker attempts to find an OR-goal to execute. If the worker fails, it sleeps for a short period, awakens and retries. If it takes too many short sleeps, it is put into a deep sleep of a much longer period (although it is believed that deep sleep does not occur for the benchmarks studied). The sleeping mechanism was installed in the original Argonne scheduler no doubt to prevent precisely the type of excessive control stack referencing that was observed here. Thus the question remained as to why the sleeping mechanism did not do its job.

The problem (as noticed by A. Ciepielewski) was that the instrumented emulator runs many times slower than the released system, but the short sleep period was set constant (a tight loop of 80 iterations). In the instrumented system, 80 iterations is proportionally too short, and must be scaled by the slowdown in emulation speed.

In addition, if the short sleep loop is lengthened, then it must issue `m_sync()`s to continue synchronizing cache simulators in other PEs (see Section 5.3). The modified short sleep code is in fact a nested loop wherein the inner loop of 80 iterations finishes with a single `m_sync()`. The outer loop is used to scale the sleep.

The short sleep modification reduced the control stack read count significantly. Still, tuning the outer-loop of the short sleep to give optimal performance, or even fair performance on all benchmarks, is difficult. Even for the non-instrumented system, short sleep time can be tuned to increase the performance of a given benchmark. We hypothesize that system performance increases with increasing short sleep time, and then decreases. To determine how sharp the performance peak is and how it varies for various benchmarks, sensitivity analysis was performed.

Table 4 lists the results of a group of 26 sensitivity experiments. Two benchmarks were measured: **Pascal** and **MBqueen**. These programs represent the extremes in availability of easy-to-exploit parallelism. Benchmark input data size, number of PEs, and short sleep time were all altered in the experiments.

By increasing the short-sleep time, idling workers disturb the system less often and make less memory requests, checking for work. Of course if the sleep time becomes too great, the program runs slower because workers are lethargic about finding new work. We see this occurring in **Pascal**. Note however that even if real-time execution increases after a certain point, the number of NODE references continues to decrease (because the workers are checking up less often). Thus it is difficult to determine exactly what is the most realistic sleep-time, i.e., where the maximum speedup is attained.

One way to do this is to compare the real-time execution of the cache instrumented system. When this execution time is minimal, one can assume that speedup (in a corresponding non-instrumented system) is maximal. Thus the memory statistics at that point are accurate. As

mentioned early, note however that even in the non-instrumented Aurora system, sleep time is not optimal for all programs. In other words, for benchmarks will little parallelism, sleep-time is probably not optimally adjusted to attain maximum speedup.

Pascal(50) running on four PEs does not obey the characteristics previously seen for 1-2 PEs. **MBqueen** on eight PEs also shows intolerance of a large sleep time. In general, **MBqueen**, and all programs with sufficient parallelism, are not significantly affected by the sleep time. Also, programs running on eight PEs appear to be little affected by large sleep times. In the measurements presented in later sections of this paper, a sleep time of one (80 iterations) was used for all benchmarks with sufficient parallelism and/or running on eight PEs. For benchmarks with insufficient parallelism (**Pascal** and **Semigroup**), a sleep time of 100 was used in conjunction with two PEs. This combination represents the best conditions under which to run these two troublesome programs.

6.2.4 Direct Write to Control Stack

The current instrumented Aurora system does *not* implement direct write refereuces to the control stack. Currently, the cache simulator implements direct write only for stacks that grow with increasing addresses ("positive stacks"). Unfortunately, the Aurora control stack is a "negative stack." One possible scheme, due to A. Matsumoto, to fool the simulator into correctly treating direct writes to the control stack, is to pass the ones-complement of the stack address. This has not yet been attempted. It is anticipated that direct write will help reduce the control stack bandwidth requirement significantly.

7 Timings and High-Level Characteristics

In this section real-parallel execution timings and speedups are presented for both the Aurora and KL1 systems. In addition, a brief summary of high level statistics is given. The Aurora and KL1 systems were calibrated for timing tests with the **KKqueen** program (see Appendix B.5.2). This program was chosen because it can be translated *directly* into Prolog. The program is also superior to the usual calibration programs, **append** or **nrev**, because it is (slightly) more complex. Single processor execution on the Sequent Symmetry [1] gave 13.16 seconds for Aurora and 13.00 seconds for KL1. This is less than a 2% difference. It can safely be assumed therefore that both systems are performing simple indexing and simple determinate computation equally well.

Table 5 gives the raw timings, relative and absolute speedups of the benchmarks.[2] Note

[1] A 12 processor Symmetry utilizing Intel 80386 CPUs. Each CPU has a 64 Kbyte write-through cache.

benchmark	PE	sleep	sec	R	R/R(1)	LR	LR/LR(1)
Pascal(30)	1	1	94	107932	1.00	413	1.00
Pascal(30)	2	1	309	2137064	19.80	39466	95.56
Pascal(30)	2	10	120	363940	3.37	8857	21.45
Pascal(30)	2	40	95	215203	1.99	6245	15.12
Pascal(30)	2	80	92	188391	1.74	5493	13.30
Pascal(30)	2	160	92	175648	1.63	5217	12.63
Pascal(30)	2	240	91	171594	1.60	5155	12.48
Pascal(50)	1	1	317	239912	1.00	833	1.00
Pascal(50)	2	1	1648	12155127	50.66	117397	140.93
Pascal(50)	2	10	400	1464906	6.11	13603	16.33
Pascal(50)	2	160	305	399601	1.67	6890	8.27
Pascal(50)	2	200	295	380469	1.59	6771	8.13
Pascal(50)	2	240	290	369551	1.54	6674	8.01
Pascal(50)	2	320	296	358663	1.49	6520	7.83
Pascal(50)	4	10	1414	6542753	27.27	105196	126.29
Pascal(50)	4	80	1512	6127034	25.54	102222	122.72
Pascal(50)	4	160	1003	6704762	27.95	112704	135.30
Pascal(50)	4	240	981	7011234	29.22	118461	142.21
Pascal(50)	4	500	1193	8651592	36.06	145455	174.62
MBqueen(8)	1	1	116	116741	1.00	77	1.00
MBqueen(8)	2	1	119	238139	2.04	7623	99.00
MBqueen(8)	2	100	111	166020	1.42	4754	61.74
MBqueen(8)	2	200	113	168473	1.44	4974	64.60
MBqueen(8)	8	1	93	219218	1.88	9963	129.39
MBqueen(8)	8	40	98	243806	2.09	12199	158.43
MBqueen(8)	8	100	98	250144	2.14	12082	156.91

Table 4: Short Sleep Time Sensitivity Analysis

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	KL1	RAW SECONDS						RELATIVE SPEEDUP					ABSOLUTE SPEEDUP			
2	benchmark	sequential	1 PE	2 PE	4 PE	8 PE		1 PE	2 PE	4 PE	8 PE		1 PE	2 PE	4 PE	8 PE
3	Triangle	244.0	286.6	147.7	81.5	49.3		1.00	1.94	3.52	5.81		0.85	1.65	2.99	4.95
4	Semigroup	395.2	416.7	222.0	140.9	87.5		1.00	1.88	2.96	4.76		0.95	1.78	2.80	4.52
5	Puzzle	322.6	359.6	181.2	94.0	55.3		1.00	1.98	3.83	6.50		0.90	1.78	3.43	5.83
6	Pascal	87.7	101.6	51.4	27.8	16.6		1.00	1.98	3.65	6.12		0.86	1.71	3.15	5.28
7	AQueen	169.9	185.0	94.3	49.5	27.3		1.00	1.96	3.74	6.78		0.92	1.80	3.43	6.22
8	KKqueen	278.9	302.2	152.9	77.9	41.5		1.00	1.98	3.88	7.28		0.92	1.82	3.58	6.72
9	KUqueen	308.4	332.8	168.3	86.2	45.6		1.00	1.98	3.86	7.30		0.93	1.83	3.58	6.76
10	E(all)							1.00	1.95	3.54	5.99		0.90	1.74	3.16	5.36
11																
12																
13	Prolog	RAW SECONDS						RELATIVE SPEEDUP					ABSOLUTE SPEEDUP			
14	benchmark	SICStus	1 PE	2 PE	4 PE	8 PE		1 PE	2 PE	4 PE	8 PE		1 PE	2 PE	4 PE	8 PE
15	Triangle	58.3	91.8	45.5	23.2	12.0		1.00	2.02	3.96	7.65		0.64	1.28	2.51	4.86
16	Semigroup	41.0	59.6	31.5	24.1	18.6		1.00	1.89	2.47	3.20		0.69	1.30	1.70	2.20
17	Puzzle	30.1	47.7	23.7	12.1	6.3		1.00	2.01	3.94	7.57		0.63	1.27	2.49	4.78
18	Pascal	64.9	84.5	56.7	47.2	41.7		1.00	1.49	1.79	2.03		0.77	1.14	1.38	1.56
19	HKqueen	59.6	80.3	42.0	23.3	14.3		1.00	1.91	3.45	5.62		0.74	1.42	2.56	4.17
20	MBqueen	110.0	142.4	73.1	38.6	21.5		1.00	1.95	3.69	6.62		0.77	1.50	2.85	5.12
21	IBqueen	364.7	598.9	302.6	155.1	82.1		1.00	1.98	3.86	7.29		0.61	1.21	2.35	4.44
22	E(all)							1.00	1.86	3.12	5.21		0.69	1.28	2.13	3.51
23																
24		KL1/Prolog														
25	benchmark		1 PE	2 PE	4 PE	8 PE										
26	Triangle		3.12	3.25	3.51	4.11										
27	Semigroup		6.99	7.05	5.85	4.70										
28	Puzzle		7.54	7.65	7.77	8.78										
29	Pascal		1.20	0.91	0.59	0.40										
30	AQ/KQueen		2.30	2.25	2.12	1.91										
31	E(all)		4.23	4.22	3.97	3.96										

Table 5: Speedups of FGHC and Prolog Benchmarks

that by averages. FGHC has greater parallelism than Prolog because the poor relative speedup of **Semigroup** and **Pascal** lowers the Prolog average. Prolog also has lower absolute speedup than FGHC because SICStus (V0.6), used as the baseline for Aurora, has a more efficient compiler than Aurora. The KL1 baseline (labeled "sequential" in Table 5) uses the same compiler as does the parallel KL1 system. The KL1 baseline is essentially the same as the parallel system except that locking is removed.

At the bottom of Table 5, the ratios of KL1 to Aurora raw execution times are given. On a single PE, Aurora outperforms KL1 by a factor of 1.2–7.5. This advantage is reduced on multiple PEs. Most notable is **Pascal** in which KL1 gains an advantage via parallelism. **Semigroup** and **Queens** also illustrate the superior parallelism of KL1, but the underlying weaknesses cannot overtake Aurora. Note that the KL1 system measured here has subsequently been improved to execute about 10% faster (via compilation optimizations of fusing common instruction pairs) [43]. It is obvious from the measurements however that an improvement of *2–9 times* is necessary to become on par with Aurora. It is wrong to assume that differential is a result purely of Aurora's mature (and KL1's immature) system implementation.

Figure 6 compares the speed of the benchmarks on a single Symmetry processor. Four systems are shown: the baseline sequential systems (seq) and the parallel systems (par) running on one PE. Figure 7 compares the speed on the benchmarks on eight Symmetry processors. Figure 8 compares the relative speedups of the benchmarks on eight Symmetry processors. Figure 9 compares the absolute and relative speedups of the benchmarks on 1–8 PEs. Note again that the wide gap between absolute speedup curves of KL1 and Aurora is because of two reasons. First, the sequential baseline for Aurora (SICStus Prolog) has a far superior compiler than Aurora. Second, the sequential baseline for KL1 is the exact same KL1 system modulo locking. Thus realistically, KL1 absolute speedup should be lower and Aurora absolute speedup should be higher. Note further that in terms of relative speedup, the different between the systems is less than their distance from ideal speedup on eight PEs.

Table 6 presents some results from the high-level instrumented emulators. Some of the data presented in Table 3 is repeated here. In addition, references are broken down into instruction and data. Statistics calculated are instructions per reduction, instructions per procedure entry, suspensions per reduction, instruction references per instruction, and data references per instruction.

For KL1, even in moderately suspending programs, suspensions compose less than 10% of all procedure entries. Thus instructions per reduction and instructions per entry are the same.

[12] In this and all other summary data, the means (`E()`) and standard deviations (`sd()`) are calculated from the five main benchmarks of this study: **Triangle**, **Semigroup**, **Puzzle**, **Pascal**, and **HKqueen** (for KL1, **AOqueen**). Each benchmark is given equal weight in the summary calculations. Other benchmarks are *not* included in the summary statistics.

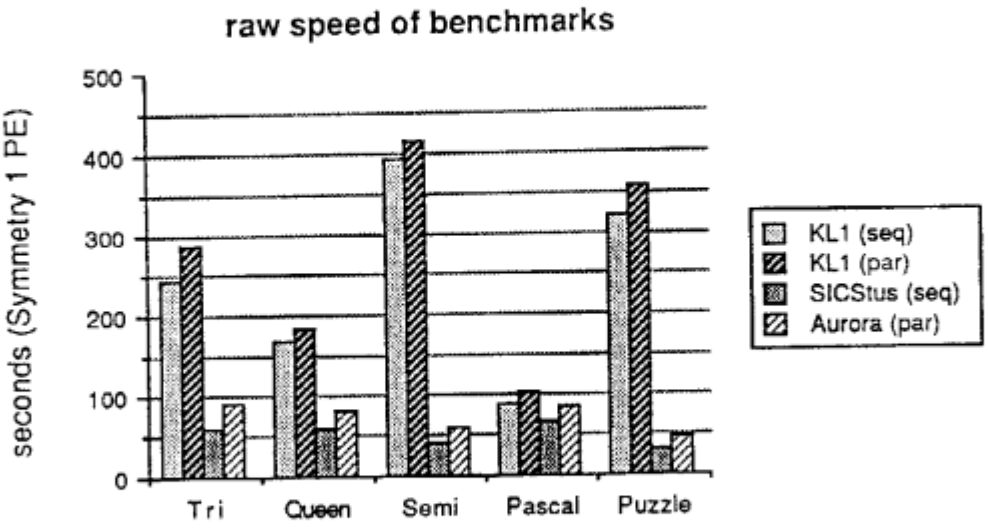


Figure 6: Raw Speed of Benchmarks on One PE

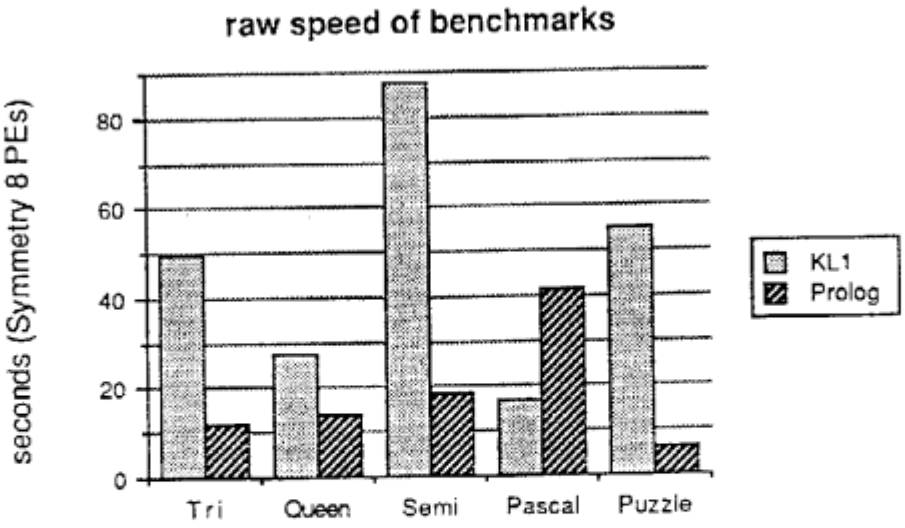


Figure 7: Raw Speed of Benchmarks on Eight PEs

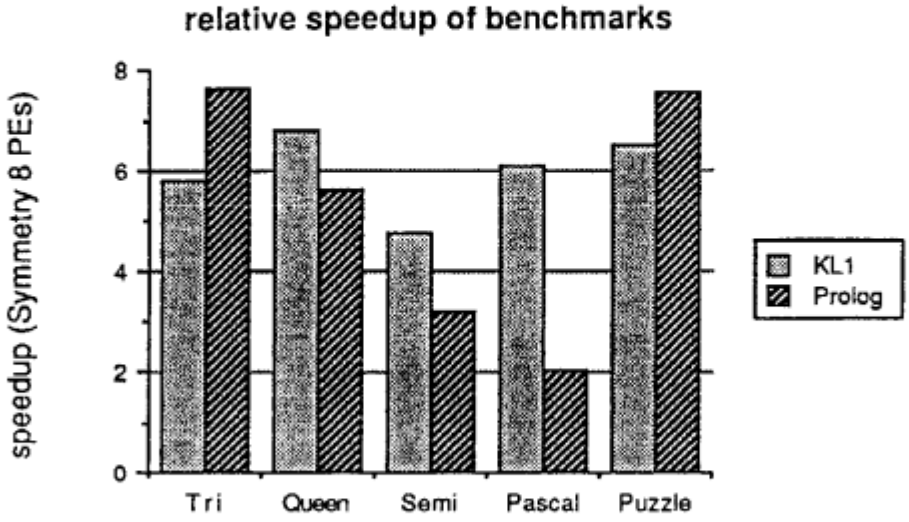


Figure 8: Relative Speedup of Benchmarks on Eight PEs

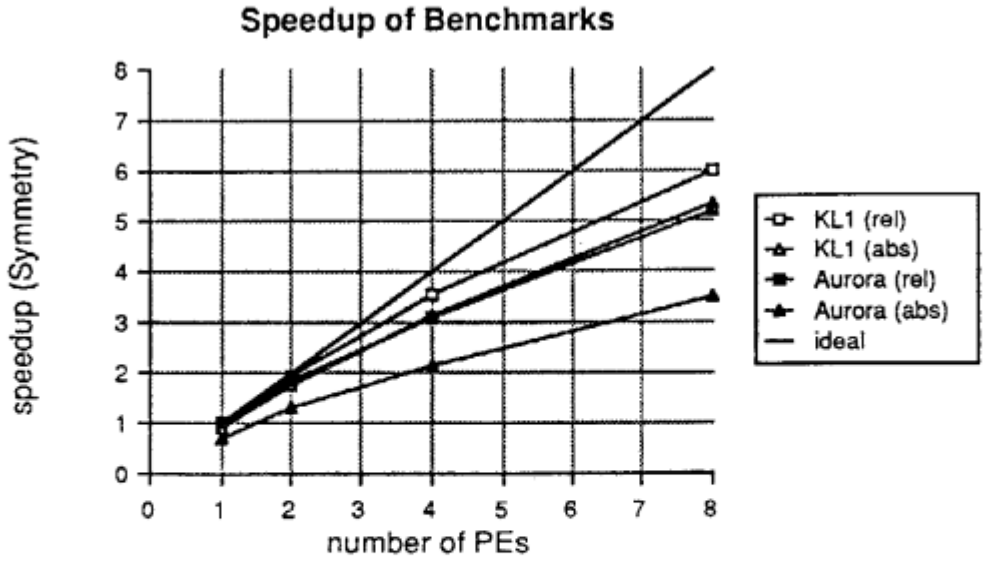


Figure 9: Absolute and Relative Speedups on 1-8 PEs

	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2													
3	KL1												
4	benchmark	Instructions	reductions	suspensions	entries	total ref	Instr ref	data ref	instr/red	instr/entry	susp/red	I-ref/instr	D-ref/instr
5	Triangle	13021727	666233	1	666234	28153917	13371797	14782120	19.55	19.55	0.00	1.03	1.14
6	Semigroup	4776416	268820	23487	292307	25069935	4793933	20276002	17.78	16.35	0.09	1.00	4.24
7	Puzzle	15606324	849539	3069	852608	29245653	16089662	13155991	18.37	18.30	0.00	1.03	0.84
8	Pascal	5018087	302432	17681	320113	9992747	5170218	4822529	16.59	15.68	0.06	1.03	0.96
9	AQuinn	10031255	333029	28865	361894	17302016	10438425	6863591	30.12	27.72	0.09	1.04	0.68
10	KKqueen	17006266	873342	77	873419	25977292	17509190	8468102	19.47	19.47	0.00	1.03	0.50
11	KUqueen	17064995	1026031	111	1026142	29398640	17386121	12012519	16.63	16.63	0.00	1.02	0.70
12	mean								20.47	19.50	0.05	1.03	1.45
13	std dev								4.92	4.33	0.04	0.01	1.41
14													
15	Prolog												
16	benchmark	Instructions	reductions	backtracks	entries	total ref	Instr ref	data ref	Instr/red	instr/entry	back/red	I-ref/instr	D-ref/instr
17	Triangle	5497442	33595	553442	587037	20927273	5837437	15289836	163.64	9.36	16.47	1.03	2.78
18	Semigroup	1928042	126151	27404	153555	11401973	3641310	7760663	15.28	12.56	0.22	1.89	4.03
19	Puzzle	1926154	61800	83306	145106	10302154	2184800	8117354	31.17	13.27	1.35	1.13	4.21
20	Pascal	2387952	197217	70059	267276	27739181	3493929	24245252	12.11	8.93	0.36	1.46	10.15
21	HKqueen	2303660	189932	144850	334782	14843223	3029128	11814095	12.13	6.88	0.76	1.31	5.13
22	MEqueen	8421586	662748	110031	772779	21898187	12967787	8930400	12.71	10.90	0.17	1.54	1.06
23	mean								46.87	10.20	3.83	1.37	5.26
24	std dev								50.81	2.38	6.33	0.30	2.56

Table 6: High-level Characteristics of Benchmarks

The average instructions per reduction of 20 has little variance among the benchmarks.

For Prolog, many programs have a significant amount of backtracking. Compared to KL1, the instructions per entry is lower because clause selection in KL1 counts as only one procedure entry, whereas in Prolog, shallow backtracking may count as many procedure entries. Prolog instructions per reduction is higher only for two programs: **Triangle** and **Puzzle**. For each of these programs, the KL1 code executes many more (smaller) procedures to simulate all-solutions search. The **Queens** programs also perform all-solutions search, but the overhead is not so great because the search tree is very simple.

WAM has 0.70 instruction references per instruction and 2.32 data references per instruction as reported by Tick [56]. KL1 has 1.03 instruction references per instruction and 1.41 data references per instruction. Aurora has 1.37 instruction references per instruction and 5.26 data references per instruction. The high instruction references per instruction is mainly because the WAM measured by Tick used real byte-code formats, whereas the systems discussed here have all instructions on word boundaries. Data referencing characteristics are more interesting.

Both KL1 and Aurora have high variances for data reference counts due to **Semigroup** and **Puzzle** respectively. Nonetheless, in general we can say that the KL1 instruction set has weaker potency than the WAM because it does not implement backtracking. Suspension referencing of course increases this statistic with respect to Prolog: however, as previously shown, the benchmarks studied do little synchronization. One exception is **Semigroup**, with 0.09 suspensions per reduction, and 4.24 data references per instruction. This is significantly higher than any of the other benchmarks measured. **AOqueen** also has a high suspension ratio, but still retains low data references per instruction.

On the other hand, Aurora displays significantly higher data references per instruction than the standard WAM. This however is not due to increased potency because both languages are Prolog. The benchmarks studied here do a significant amount of complex pattern matching and backtracking, thus increasing data referencing above that of the more "realistic" programs studied by Tick (two compilers, **CHAT**, and a theorem prover). **Puzzle** is the most intensive program of the group in this respect, with 10.15 data references per instruction. In addition, the overheads of scheduling also increase data referencing.

8 Memory Referencing Characteristics

In this section, the memory and bus usage characteristics of Aurora and KL1 are described.

8.1 Memory References

Tables 7 and 8 give the raw simulation memory referencing profiles of the benchmarks measured in this study. For each benchmark, the memory reference count is broken down by reference type and area. For each row and column, percentages are displayed. Note that for **Semigroup** and **Pascal** in Aurora, two PE emulation statistics were used throughout. This is because these benchmarks display excessive scheduler behavior on eight PEs that prevents viewing a "normal" execution profile.

The raw data is summarized in a series of tables and figures. Tables 9 and 10 give the means and standard deviations of the memory references broken down by storage area and memory operation. Figures 10 and 11 summarize this data in the form of pie charts. Each figure shows summary statistics for all references, and for data references only.

The KL1 and Aurora referencing characteristics are primarily skewed by the large percentage of KL1 instruction references, 47% of all references compared to Aurora's 27%. Aurora has on average 2.76 data references per instruction reference. KL1 has on average 1.14 data references per instruction reference. Prolog was measured even higher at 3.46 data references per instruction reference [56]. The differences are due in part to the instruction formats, parallel overheads, and language *potency*. Aurora is the most efficiently encoded instruction set (more efficienty than the Prolog system measured by Tick). In addition, Prolog is more semantically powerful (potent) than FGHC, and its corresponding architecture is also more powerful (i.e., more work is performed by each individual instruction, on average). However, Aurora has overheads of parallel execution: scheduler work is counted as data references with no instruction fetches made. Overall, Aurora therefore falls inbetween Prolog and KL1 in data references per instruction reference.

The over 2:1 ratio between Aurora and KL1 for this statistic is also felt in the skewed read reference counts. Even given this bias however, KL1 and Aurora have almost the same percentage of reads. When instruction references are removed from the statistics, Aurora has 70% reads to KL1's 61%. In both cases, the read:write ratio is higher than Prolog (53% read data references [56]). The reason for this is not because the architectures are more efficient than Prolog, but because each has scheduling and synchronization overheads that require many reads. In Aurora, the tree-walker generates control stack read traffic. In KL1, we measure 8.5% read data references to lock variables (during dereferencing and/or binding).

The direct write optimization was utilized in 9.5% of KL1 data references and 4.6% of Aurora data rereferences. This statistic is biased unfairly towards KL1 because the Aurora control stack was *not* instrumented with direct write operations, although it could have been (see Section 6.2.4). The statistics presented show no percentages for optimizations used to implemented one-time write-read buffers (see Section 6.1.6). KL1 can utilize these optimizations quite effectively

	A	B	C	D	E	F	G	H	I	J	K	L
1	Triangle	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
2	R	13371797	924237	4060064	72	0	437751	0	5422124	18793921	66.75	36.68
3	W	0	61375	4059987	71	0	437771	0	4559204	4559204	16.19	30.84
4	DW	0	970384	0	0	0	0	0	970384	970384	3.45	6.56
5	LR	0	1691285	0	0	0	206066	0	1897351	1897351	6.74	12.84
6	UW	0	575707	0	0	0	206066	0	781773	781773	2.78	5.29
7	U	0	1151284	0	0	0	0	0	1151284	1151284	4.09	7.79
8	TOTAL	13371797	5374272	8120051	143	0	1287654	0	14782120	28153917	100.00	100.00
9		47.50	19.09	28.84	0.00	0.00	4.57	0.00	52.50	100.00		
10												
11	Semigroup	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
12	R	4793933	18394554	329437	6762	0	42891	0	18773644	23567577	94.01	92.59
13	W	0	5093	321742	6761	0	44195	0	377791	377791	1.51	1.86
14	DW	0	489863	0	0	0	0	0	489863	489863	1.95	2.42
15	LR	0	295475	0	0	0	20628	0	316103	316103	1.26	1.56
16	UW	0	294546	0	0	0	20628	0	315174	315174	1.26	1.55
17	U	0	3427	0	0	0	0	0	3427	3427	0.01	0.02
18	TOTAL	4793933	19482958	651179	13523	0	128342	0	20276002	25069935	100.00	100.00
19		19.12	77.71	2.60	0.05	0.00	0.51	0.00	80.88	99.99		
20												
21	Puzzle	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
22	R	16089662	2795179	2047966	8604	0	65086	0	4916835	21006497	71.83	37.37
23	W	0	11171	2025186	8602	0	65144	0	2110103	2110103	7.22	16.04
24	DW	0	2724446	0	0	0	0	0	2724446	2724446	9.32	20.71
25	LR	0	1523443	0	0	0	30847	0	1554290	1554290	5.31	11.81
26	UW	0	890919	0	0	0	30847	0	921766	921766	3.15	7.01
27	U	0	928551	0	0	0	0	0	928551	928551	3.18	7.06
28	TOTAL	16089662	8873709	4073152	17206	0	191924	0	13155991	29245653	100.01	100.00
29		55.02	30.34	13.93	0.06	0.00	0.66	0.00	44.98	100.01		
30												
31	Pascal	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
32	R	5170218	472010	1284773	61860	0	112826	0	1911269	7081487	70.87	39.63
33	W	0	8217	1190776	59744	0	112954	0	1371691	1371691	13.73	28.44
34	DW	0	474474	0	0	0	0	0	474474	474474	4.75	9.84
35	LR	0	446313	0	4948	0	49462	0	500723	500723	5.01	10.38
36	UW	0	318233	0	4948	0	49462	0	372643	372643	3.73	7.73
37	U	0	191729	0	0	0	1	0	191730	191730	1.92	3.98
38	TOTAL	5170218	1910976	2455549	131300	0	324705	0	4822530	9992748	100.01	100.00
39		51.74	19.12	24.57	1.31	0.00	3.25	0.00	48.26	99.99		
40												
41	AQueen	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
42	R	10438425	2011846	1452233	146625	0	243667	0	3854371	14292796	82.61	56.16
43	W	0	4826	1192371	128746	0	243715	0	1569658	1569658	9.07	22.87
44	DW	0	548048	0	0	0	0	0	548048	548048	3.17	7.98
45	LR	0	318786	0	0	0	95467	0	414253	414253	2.39	6.04
46	UW	0	304468	0	0	0	95467	0	399935	399935	2.31	5.83
47	U	0	77326	0	0	0	0	0	77326	77326	0.45	1.13
48	TOTAL	10438425	3265300	2644604	275371	0	678316	0	6863591	17302016	100.00	100.01
49		60.33	18.87	15.28	1.59	0.00	3.92	0.00	39.67	99.99		
50												
51	KKqueen	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
52	R	17509190	2062291	1994179	1509	0	91333	0	4149312	21658502	83.37	49.00
53	W	0	11397	1992444	1422	0	91372	0	2096635	2096635	8.07	24.76
54	DW	0	817084	0	0	0	0	0	817084	817084	3.15	9.65
55	LR	0	573280	0	0	0	44571	0	617851	617851	2.38	7.30
56	UW	0	376572	0	0	0	44571	0	421143	421143	1.62	4.97
57	U	0	366077	0	0	0	0	0	366077	366077	1.41	4.32
58	TOTAL	17509190	4206701	3986623	2931	0	271847	0	8468102	25977292	100.00	100.00
59		67.40	16.19	15.35	0.01	0.00	1.05	0.00	32.60	100.00		
60												
61	KUqueen	INSTR	HEAP	GOAL	SUSP	META	COMM	ETC	DATA	TOTAL	% (ALL)	% (DATA)
62	R	17386121	6694415	1166883	582	0	58852	0	7920732	25306853	86.08	65.94
63	W	0	7381	1166297	582	0	58884	0	1233144	1233144	4.19	10.27
64	DW	0	1923983	0	0	0	0	0	1923983	1923983	6.54	16.02
65	LR	0	399407	0	0	0	28916	0	428323	428323	1.46	3.57
66	UW	0	155048	0	0	0	28916	0	183964	183964	0.63	1.53
67	U	0	322373	0	0	0	0	0	322373	322373	1.10	2.68
68	TOTAL	17386121	9502607	2333180	1164	0	175568	0	12012519	29398640	100.00	100.01
69		59.14	32.32	7.94	0.00	0.00	0.60	0.00	40.86	100.00		
70												

Table 7: Memory Referencing Characteristics of KL1: Raw Data

	A	B	C	D	E	F	G	H	I	J	K	L
1	Triangle	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
2	R	5637437	4578021	225489	5724748	41878	41016	1186599	11797751	17435188	83.31	77.16
3	W	0	16658	17326	761148	38123	1138899	1182083	3154237	3154237	15.07	20.63
4	DW	0	334092	0	0	0	0	0	334092	334092	1.60	2.19
5	LR	0	0	0	1878	0	0	0	1878	1878	0.01	0.01
6	UW	0	0	0	0	0	0	0	0	0	0.00	0.00
7	U	0	0	0	1878	0	0	0	1878	1878	0.01	0.01
8	TOTAL	5637437	4928771	242815	6489652	80001	1179915	2368682	15289836	20927273	100.00	100.00
9		26.94	23.55	1.16	31.01	0.38	5.64	11.32	73.06	100.00		
10	(2 PE)											
11	Semigroup	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
12	R	3638548	1740864	1084540	805858	69219	329961	66278	4076720	7715268	88.41	80.12
13	W	0	207181	107552	259968	10878	12005	55673	653237	653237	7.49	12.84
14	DW	0	340570	0	0	0	0	0	340570	340570	3.90	6.69
15	LR	0	0	0	8755	0	0	0	8755	8755	0.10	0.17
16	UW	0	0	0	0	0	0	0	0	0	0.00	0.00
17	U	0	0	0	8755	0	0	0	8755	8755	0.10	0.17
18	TOTAL	3638548	2288595	1172092	1083336	80097	341966	121951	5088037	8726585	100.00	100.00
19		41.69	26.23	13.43	12.41	0.92	3.92	1.40	58.31	100.00		
20												
21	Puzzle	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
22	R	2184800	1908806	163577	1442251	33099	601789	1133626	5283148	7467948	72.49	65.08
23	W	0	13025	12758	906573	30666	585867	1128133	2677022	2677022	25.99	32.98
24	DW	0	153204	0	0	0	0	0	153204	153204	1.49	1.89
25	LR	0	0	0	1990	0	0	0	1990	1990	0.02	0.02
26	UW	0	0	0	0	0	0	0	0	0	0.00	0.00
27	U	0	0	0	1990	0	0	0	1990	1990	0.02	0.02
28	TOTAL	2184800	2075035	176335	2352804	63765	1187656	2261759	8117354	10302154	100.00	100.00
29		21.21	20.14	1.71	22.84	0.62	11.53	21.95	78.79	100.00		
30	(2 PE)											
31	Pascal	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
32	R	3492551	2526195	1457701	2024894	217774	247806	1196047	7670417	11162968	72.45	64.38
33	W	0	108290	174153	1839243	99402	103922	1188081	3513091	3513091	22.80	29.49
34	DW	0	716908	0	0	0	0	0	716908	716908	4.65	6.02
35	LR	0	0	0	6951	0	0	0	6951	6951	0.05	0.06
36	UW	0	0	0	0	0	0	0	0	0	0.00	0.00
37	U	0	0	0	6951	0	0	0	6951	6951	0.05	0.06
38	TOTAL	3492551	3351393	1631854	3878039	317176	351728	2384128	11914318	15406869	100.00	100.00
39		22.67	21.75	10.59	25.17	2.06	2.28	15.47	77.33	100.00		
40												
41	HKqueen	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
42	R	3029128	1922845	1321129	2375784	534172	593511	704161	7451602	10480730	70.61	63.07
43	W	0	221328	108180	1923317	299285	356071	678309	3586490	3586490	24.16	30.36
44	DW	0	756659	0	0	0	0	0	756659	756659	5.10	6.40
45	LR	0	0	0	9672	0	0	0	9672	9672	0.07	0.08
46	UW	0	0	0	0	0	0	0	0	0	0.00	0.00
47	U	0	0	0	9672	0	0	0	9672	9672	0.07	0.08
48	TOTAL	3029128	2900832	1429309	4318445	833457	949582	1382470	11814095	14843223	100.00	100.00
49		20.41	19.54	9.63	29.09	5.62	6.40	9.31	79.59	100.00		
50												
51	MBqueen	INSTR	HEAP	ENV	NODE	LBA	GBA	TRAIL/ETC	DATA	TOTAL	% (ALL)	% (DATA)
52	R	12967787	2286109	1101150	1821667	349711	249020	494167	6301824	19269611	85.97	66.70
53	W	0	78448	108194	1578456	299106	153873	474119	2690196	2690196	12.00	28.47
54	DW	0	440738	0	0	0	0	0	440738	440738	1.97	4.67
55	LR	0	0	0	7414	0	0	0	7414	7414	0.03	0.08
56	UW	0	0	0	7414	0	0	0	7414	7414	0.03	0.08
57	U	0	0	0	0	0	0	0	0	0	0.00	0.00
58	TOTAL	12967787	2803295	1209344	3414951	648817	402893	968286	9447586	22415373	100.00	100.00
59		57.85	12.51	5.40	15.23	2.89	1.80	4.32	42.15	100.00		
60												

Table 8: Memory Referencing Characteristics of Aurora: Raw Data

area	$E(all)$	$\sigma(all)$	$E(data)$	$\sigma(data)$
INSTR	46.74	14.43		
DATA	53.26	14.43		
HEAP	33.03	22.77	62.02	
GOAL	17.04	9.13	31.99	
SUSP	0.60	0.70	1.13	
COMM	2.58	1.68	4.84	
operation	$E(all)$	$\sigma(all)$	$E(data)$	$\sigma(data)$
R	77.21	9.90	52.49	21.29
W	9.54	5.14	20.01	10.41
DW	4.53	2.56	9.50	6.11
LR	4.14	2.01	8.53	4.18
UW	2.65	0.83	5.48	2.15
U	1.93	1.56	4.00	3.09

Table 9: KL1 % Memory References by Area and Operation

area	$E(all)$	$\sigma(all)$	$E(data)$	$\sigma(data)$
INSTR	26.58	7.88		
DATA	73.42	7.88		
HEAP	22.24	2.43	30.30	
ENV	7.30	4.96	9.95	
NODE	24.11	6.51	32.83	
LBA	1.92	1.94	2.61	
GBA	5.95	3.13	8.11	
TRAIL/ETC	11.89	6.80	16.20	
operation	$E(all)$	$\sigma(all)$	$E(data)$	$\sigma(data)$
R	77.46	7.08	69.96	7.18
W	19.10	6.90	25.26	7.47
DW	3.35	1.52	1.61	2.14
LR	0.05	0.03	0.07	0.06
U	0.05	0.03	0.07	0.06

Table 10: Aurora % Memory References by Area and Operation

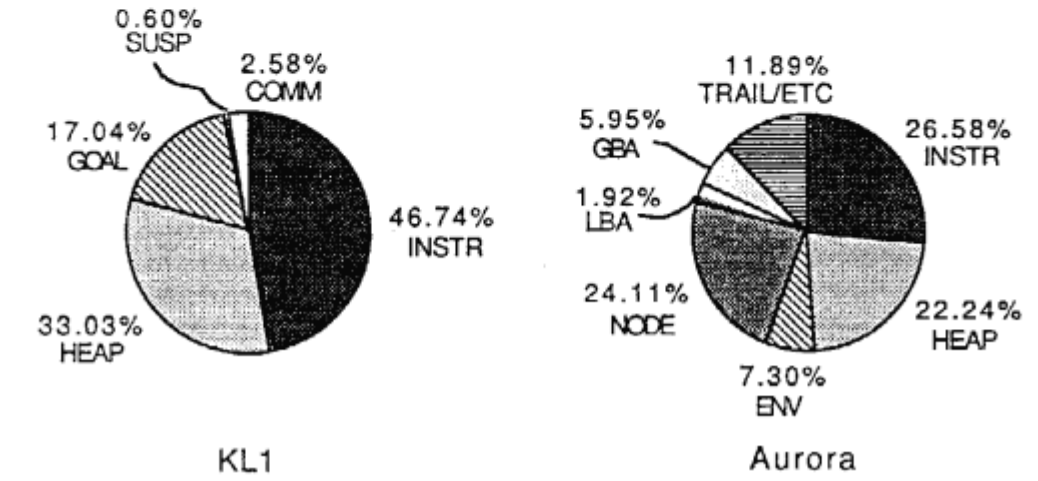
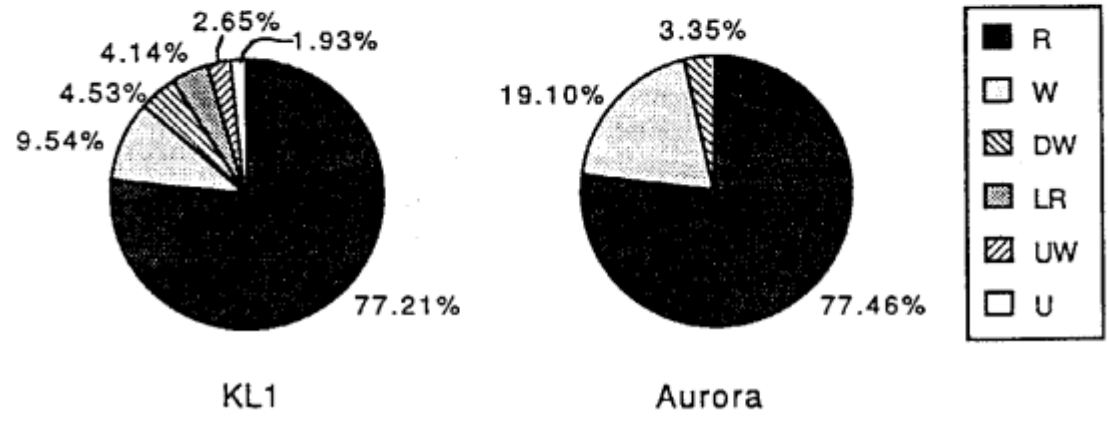
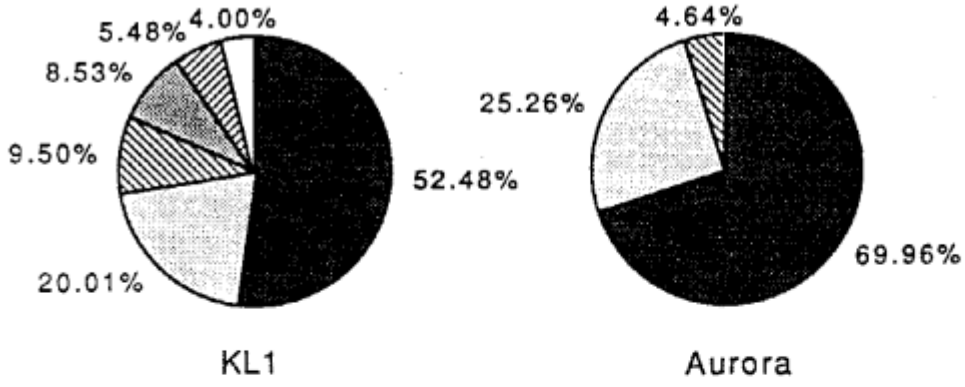


Figure 10: Memory Referencing Characteristics (by Area) of KL1 and Aurora



Percentage Memory References by Operation



Percentage Data Memory References by Operation

Figure 11: Memory Referencing Characteristics (by Operation) of KL1 and Aurora

for the goal and communication areas. However, Aurora cannot use such optimizations.

The data referencing characteristics of each architecture are now discussed in more detail. Note that these statistics are deceptive because bus traffic, the critical concern in a shared memory multiprocessor, is only indirectly related to the raw reference counts. Locality and sharing in the areas radically effects the bus traffic generated, as shown later in this section.

KL1 data referencing characteristics measured in this study differ significantly from those measured by Matsumoto for the **BUP** benchmark. Yet as shown in Section 9.1, the **BUP** statistics calibrate on both simulators. This shows that FGHC programs have vastly different characteristics and that many benchmarks need to be studied to get a fair and accurate picture of processor performance. In this study, the heap is referenced 62% on average, goals 32%, and communication 5%. Suspensions do not effect reference counts significantly.

Aurora is more complex, with a balanced mix of references to heap, environment, control (node), trail, and binding array areas. This profile is radically different than that measured for Prolog (53% control, 23% environment, 20% heap, 3% trail). The differences are explained as follows. 11% of Aurora references are devoted to the binding arrays, so these references must be factored out when compared to Prolog. In addition, trailing in Aurora requires saving both an address and value, twice the storage requirements of the Prolog trail. In addition, the Aurora compiler generates efficient code that can reduce node referencing during shallow backtracking. The Prolog statistics were gathered on a system without such optimizations. These considerations help to calibrate the two variations of the WAM; however, the rather low environment data reference count in Aurora has not yet been explained. This is again possibly due to the sophistication of the Aurora compiler.

8.2 Bus Traffic

Tables 11 and 12 give the raw simulation bus traffic profiles of the benchmarks measured in this study. For each benchmark, the percentage bus traffic is broken down by area. This raw data is redisplayed in graphic form in Figure 13. The model used to generate these measurements is a shared instruction data (I+D) cache coupled with a one word bus and eight cycle memory. This data is presented with the intention of delineating the trouble spots in each architecture. However, note that a split instruction and data cache and/or a different bus and memory model will produce different profiles for the same benchmarks. Note also that the summary statistics are calculated with Aurora **Semigroup** and **Pascal** running on two PEs.

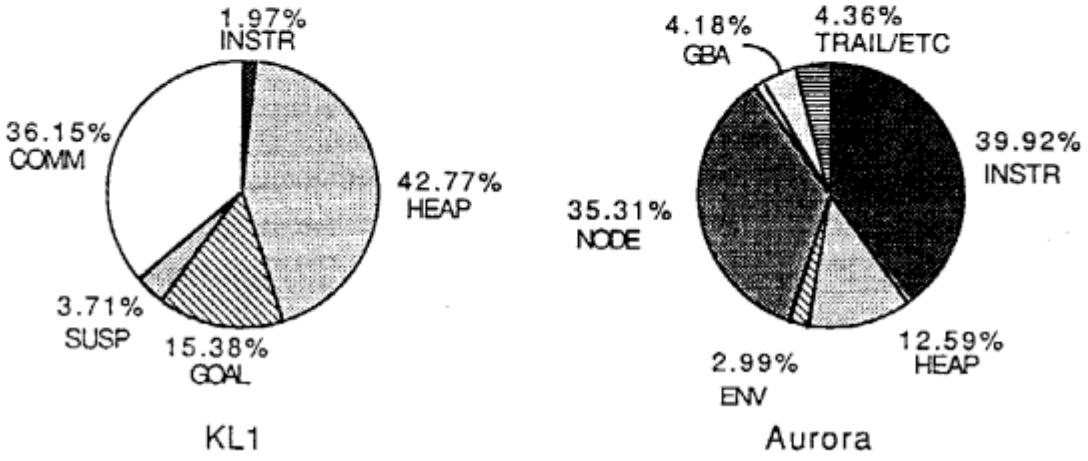
Aurora's bus traffic characteristics vary greatly with each benchmark. Instruction bus traffic varies from 15–77%. Within data traffic, heap bus traffic varies from 1–39% and node bus traffic varies from 9–58%. The other areas are more stable across the benchmarks. KL1's bus traffic characteristics also vary with each benchmark. Instruction bus traffic varies from 0.1–4.9%.

benchmark	INSTR	DATA	HEAP	GOAL	SUSP	COMM
Triangle	2.1	97.9	27.8	33.8	0.0	36.4
Semigroup	2.0	98.0	52.3	10.4	2.0	33.4
Puzzle	4.9	95.1	74.3	10.5	0.5	9.7
Pascal	0.7	99.3	32.4	15.5	6.1	45.3
AOqueen	0.1	99.9	27.2	6.7	10.0	56.0
KKqueen	0.1	99.9	45.3	25.5	0.2	29.0
KUqueen	0.1	99.9	72.0	13.3	0.1	14.6
$E(all)$	2.0	98.0	42.8	15.4	3.7	36.2
$\sigma(all)$	1.7	1.7	18.2	9.6	3.8	15.4
$E(data)$			43.6	15.7	3.8	36.9

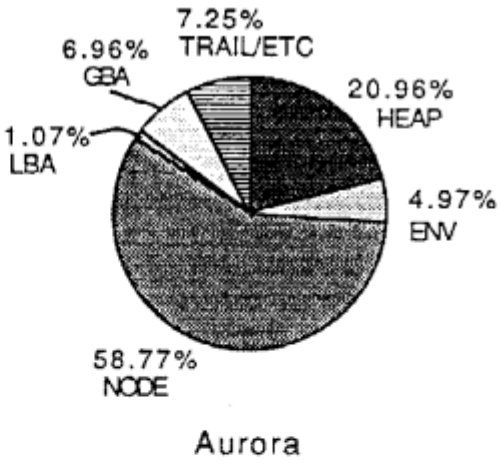
Table 11: KL1 % Bus Traffic by Area

benchmark	INSTR	DATA	HEAP	ENV	NODE	LBA	GBA	TRAIL
Triangle	47.9	52.1	1.5	3.8	38.4	0.7	2.2	5.6
Puzzle	34.2	65.8	6.9	2.9	42.9	0.6	5.0	7.5
Semigroup	4.9	95.2	10.7	1.8	75.0	0.6	4.0	3.0
Semigroup*	15.4	84.6	39.4	4.7	27.5	1.0	7.4	4.5
Pascal	14.6	85.4	2.9	1.2	78.6	0.3	1.3	1.0
Pascal*	76.6	23.5	5.2	2.1	9.4	0.6	4.1	2.0
HKqueen	25.6	74.4	1.4	1.4	58.3	0.3	2.3	2.1
MBqueen	29.1	70.9	9.4	1.3	57.2	0.1	1.5	1.2
$E(all)$	60.1	39.9	12.6	3.0	35.3	0.7	4.2	4.4
$\sigma(all)$	21.2	21.2	13.7	1.2	16.3	0.2	2.0	2.1
$E(data)$			21.0	5.0	58.8	1.1	7.0	7.3

Table 12: Aurora % Bus Traffic by Area (* = 2 PEs)



Percentage Bus Cycles by Area



Percentage Bus Cycles by Data Area

Table 13: Bus Traffic Characteristics (by Area) of KL1 and Aurora

Within data traffic, suspension bus traffic varies from 0–10% and communication bus traffic varies from 10–56%.

The most significant difference between the architectures is the instruction bus traffic. Aurora generates a great deal of instruction bus traffic, whereas KL1 has almost none. It would therefore appear than KL1 has superior code locality. This may be due to the fact that KL1 does not backtrack, and in these benchmarks, does little synchronization. Thus execution is determinate and jumps are infrequent. With few jumps, the prefetch effect of four word cache blocks gives KL1 a very low instruction miss ratio. On the other hand, the Prolog programs are heavily backtracking. This however cannot explain why **Pascal** instruction traffic differs by a factor of 67, whereas Aurora makes only 68% of KL1's instruction requests!

Another interesting statistic is that Aurora heap bus traffic is on average 21% of all data bus traffic, compared to 44% for KL1. Similarly, Aurora environment bus traffic is on average 5% of all data bus traffic, compared to 16% for KL1. These results indicates in part that Prolog's stack-group storage management has superior spatial locality to KL1's heap-based storage management. The results also derive from Aurora's high scheduling overhead: the control stack alone generates 59% of all data bus traffic. The binding arrays and trail account for 15% more. One can roughly compare this to KL1's suspension and communication traffic of 41%. Thus the statistics reenforce the hypotheses that committed-choice languages require simpler management than do non-committed-choice languages (because there is no backtracking, nor multiple bindings of the same variable), but that committed-choice languages have less data locality because of the necessity for heap-based management. Specifically, OR-parallel Prolog requires high control-stack bus bandwidth because the individual PEs are walking around the OR-tree, executing the program. AND-parallel FGHC requires heap-based storage management because procedure environments cannot be stored effectively on a true stack.

Looking at Aurora, **Semigroup** and **Pascal** have different characteristics for two and eight PEs. **Semigroup** is data intensive in either case. Although node bus traffic decreases to 27.5% of all data bus traffic on two PEs, 84.6% of all bus cycles are spent on data transfers. **Pascal** is *not* data intensive, as is shown by the decrease of node bus traffic to only 9.4% of all data bus traffic on two PEs. As a result, instruction traffic becomes significant on two PEs: 76.6% of all bus cycles. These characteristics can be seen in the benchmark code. **Semigroup** manipulates lists of 40 integers whereas **Pascal** manipulates varying size lists of at most six integers.

9 Cache Performance

In this section, cache simulation results for the Aurora and KL1 systems are presented. First, the (in)accuracies of the cache model used are described. In the case of KL1, the simulator

is calibrated against an earlier simulator measured by Matsumoto [35]. Second, the cache measurements, in terms of miss and bus traffic ratios are presented and analyzed.

9.1 Calibration

In this section, calibration with Matsumoto's results using a pseudo-parallel simulator [35] are presented. Calibration of the Aurora system is not possible because there have been no previous studies of Aurora's cache behavior.

Table 14 shows the percentages of the memory references and bus cycles for each area in the KL1 abstract machine, for both the new and old (pseudo-parallel) simulators. The benchmark measured is **BUP**. All measurements presented are for a no-indexing version of **BUP** (because the old compiler did not have indexing). In addition, the old statistics are calibrated by removing all references to meta-counts (the new simulator does not count meta-control).

Table 14 shows that the new and old simulators are closely calibrated. There are a few significant differences however. The new simulator performs fewer suspensions than the old simulator—this is no doubt due to timing differences. The lower suspension count of the new simulator is felt to be more accurate than the old simulator. The decrease in suspension count affects the other statistics, for instance the decrease in heap references.

The old simulator uses direct write (DW) commands for the goal and communication areas. The new simulator does not implement this optimization. Note that although the old simulator made 32% more goal references, it used 63% fewer bus cycles. Similarly, the old simulator made 10% more communication references, but used 37% fewer bus cycles. In the case of communication, the old simulator is inaccurate because it does not lock and unlock the communication area before sending a message, as is necessary in a real-parallel system. Matsumoto [35] reported that cache optimizations to allow direct writes to the goal and communication areas reduced total bus traffic by 6%. We see here that since suspensions are actually lower than measured by Matsumoto, and assuming that meta-control can be implemented at low cost, the relative savings afforded by direct write to the goal and communication areas is far greater than 6%. Note that whereas in the old simulation, heap referencing accounted for the most bus traffic, in the new simulation, communication referencing is the culprit.

9.2 Results

The majority of the plots presented in this section have increasing cache size on the X-axis, and increasing miss (or bus traffic) ratio on the Y-axis. Unless otherwise stated, all simulations were run with eight PEs, a cache block size of four words, four-way set associativity, and write allocation (i.e., if a write request misses in the cache, the target line is allocated in the cache).

	references			bus cycles		
area	old:new	% new	% old	old:new	% new	% old
INST	1.21	52.8	51.4	1.75	13.9	16.8
DATA	1.38	47.2	48.6	1.71	86.1	83.2
HEAP	1.09	17.6	15.5	2.46	25.8	43.6
GOAL	1.32	27.0	28.9	0.37	21.1	5.4
SUSP	4.68	0.6	2.2	5.74	4.9	19.5
COMM	1.10	2.0	2.1	0.63	33.9	14.6

Table 14: Calibration of KL1 Simulators Using **BUP**

Simulations marked with an asterisk (*) were run with two PEs.

The cache sizes simulated are 32, 64, 128, 256, and 512 columns, corresponding to data areas of 512, 1024, 2048, 4096 and 8192 words. In the plots however, the plotted cache sizes are as calculated in Section 5.3.3. This calculation assumes a 5 byte data word and accounts for directory size. Through this discussion, cache sizes are distinguished by their word size, e.g., "a 2048 word cache."

The bus traffic ratio (BTR) plots presented all assume a two word bus and eight cycle memory access time. Figure 12 shows the effect of these parameters on the BTR. The X-axis coordinates represent the ten models considered. For example the third coordinate is 2.6 representing a two word bus and a six cycle memory access. Recall from Section 5.3.3 that zero access time models imply that bus operations can be overlapped. Two typical benchmarks are given in Figure 12. For each, bus traffic increases as the models degrade. With the standard non-overlapped bus model, bus traffic is only weakly dependent on memory access time. The introduction of an overlapped bus offers the most reduction of traffic. However, it is clearly more beneficial to double the bus width than speedup the memory or implement a complex overlap manager.

The main plots illustrate the following experimental space: two architectures (Aurora and KL1), five benchmarks, five cache sizes, two cache types (data-only and instruction+data), two statistics (miss ratio and bus traffic ratio). In addition, plots are given illustrating a subset of the following extensions to this space: other benchmarks, two processor configurations (two and eight PEs), ten system models (varying bus width and memory access time).

The cache simulations performed for this study are "empirical" in a stronger sense of the word than standard uniprocessor cache simulations, or even psuedo-parallel multiprocessor cache simulations. Here a real-parallel emulator and cache simulator were run, and so the statistics include the probabilistic effects of timing. Therefore occasionally the data appears "to go in the wrong direction" – this should indicate the variances involved. In other words, if

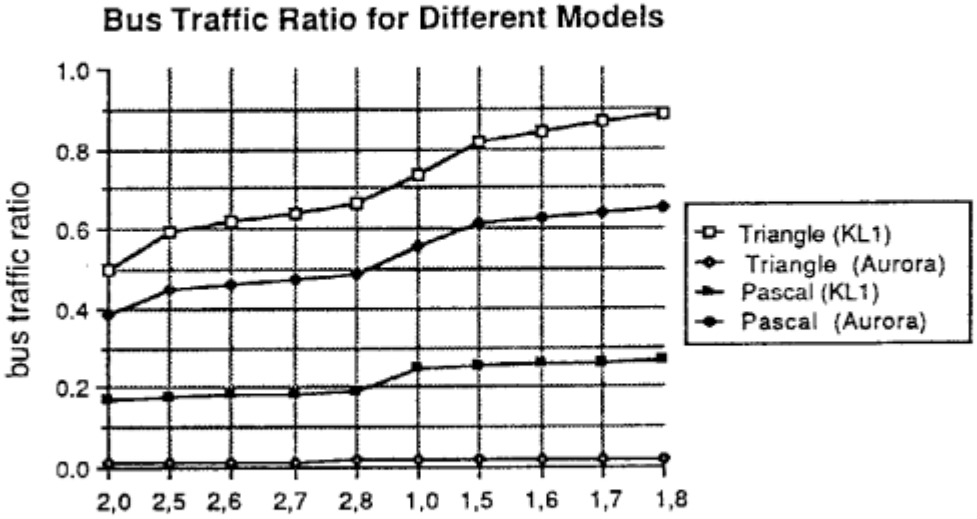


Figure 12: Comparison of Bus Traffic for Different System Models

a large cache performs worse than a smaller cache for some experiment, it may well be that the timing of the two simulations was such that the smaller cache accidentally made better scheduling decisions. This problem is especially severe for timing critical FGHC programs, such as **AOqueen**. **AOqueen** is a drastic example where the cache simulation interfered with the timing to the extent of significantly altering the number of suspensions, affecting both high-level and low-level statistics.

Examining I+D caches first (Tables 13 and 14), we find Aurora achieving lower miss and bus traffic ratios than KL1 as cache size increases. The KL1 curves (except for **Triangle**) flatten out almost precisely at 10^5 bits (2048 word cache), whereas Aurora continues to improve. Note that different benchmarks have drastically different behavior on the two systems, as we would expect from the high-level results in previous sections. For example, **Semigroup** and **Pascal** display higher bus traffic for a 2048 word cache in Aurora (even on two PEs) than in KL1. However, at 8192 words, Aurora can achieve lower bus traffic on two PEs, but still not on eight PEs (due to scheduling bandwidth). For **Triangle**, the roles are reversed, and Aurora has consistently lower bus traffic. **Puzzle** and **Queens** show almost equal performance for both systems on 512 word caches, but Aurora improves more rapidly with increasing cache size.

In general, KL1 performance is "flat," indicative of an architecture with a ever changing working set. KL1 monotonically walks through memory, referencing fresh areas on the way (until GC is incurred). Still, reasonable cache performance is achieved because the execution

mechanism rereferences the same area frequently, as the walk through memory proceeds. Aurora performance is more "classical," i.e., bus traffic and miss ratio continue to decrease gracefully with cache size. Most of the benchmarks have their entire working sets captured in caches of 2048 words and larger.

Comparing the D-cache to the I+D-cache statistics, we find that Aurora and KL1 have opposite results. Aurora D-cache performance is *better* for all the benchmarks except **Semigroup**, than its I+D cache performance. KL1 performance is exactly opposite. This confirms the results seen in Section 8.2 that Aurora instruction referencing has lower locality than KL1 instruction referencing, and visa-versa for data referencing. Again, these characteristics can be explained by Aurora's more efficient stack-based storage model and its more "jumpy" code style.

Table 15 shows the two and eight PE versions of **Semigroup** and **Pascal**. These graphs don't say much, simply that overall, the extra traffic induced by the scheduler has a constant effect for all cache sizes. Table 16 shows an in-depth look at the miss ratios for the **Queens** benchmarks running on eight PEs (bus traffic ratios are similar). The relative performance gap between algorithms is easily viewed. For KL1, the beneficial effect of adding instructions to the cache is seen. For Aurora, both data-only and I+D caches have the same performance. **AOqueen** displays unstable behavior because of timing sensitivity, but in general, all the KL1 programs get no improvement with increasing cache size. This is because the KL1 working sets are constantly changing. The Aurora curves show the behavior of larger caches capturing the working set.

10 Conclusions

This study attempts to quantify the performance differences between committed-choice and non-committed-choice parallel logic programming language architectures. Specifically, the Aurora OR-parallel Prolog system is compared to the KL1 AND-parallel FGHC system for equivalent benchmark programs. Because the systems differ in both the type of parallelism exploited *and* the facility for non-determinate execution, separation of effects is difficult to analyze. Added are the differences of scheduling methods, garbage collection, and various other system support. This study claims not to convincingly analyze each effect in separation, but does present data that can help designers understand architecture tradeoffs. The field of parallel architectures is not so very young, but the field of parallel high-level language architectures (e.g., Lisp and Prolog-based languages) is almost infantile. Therefore experience with these types of architectures is limited, and accurate measurements of these experiences is even more limited. This paper presents the first detailed performance characteristics of parallel logic programming

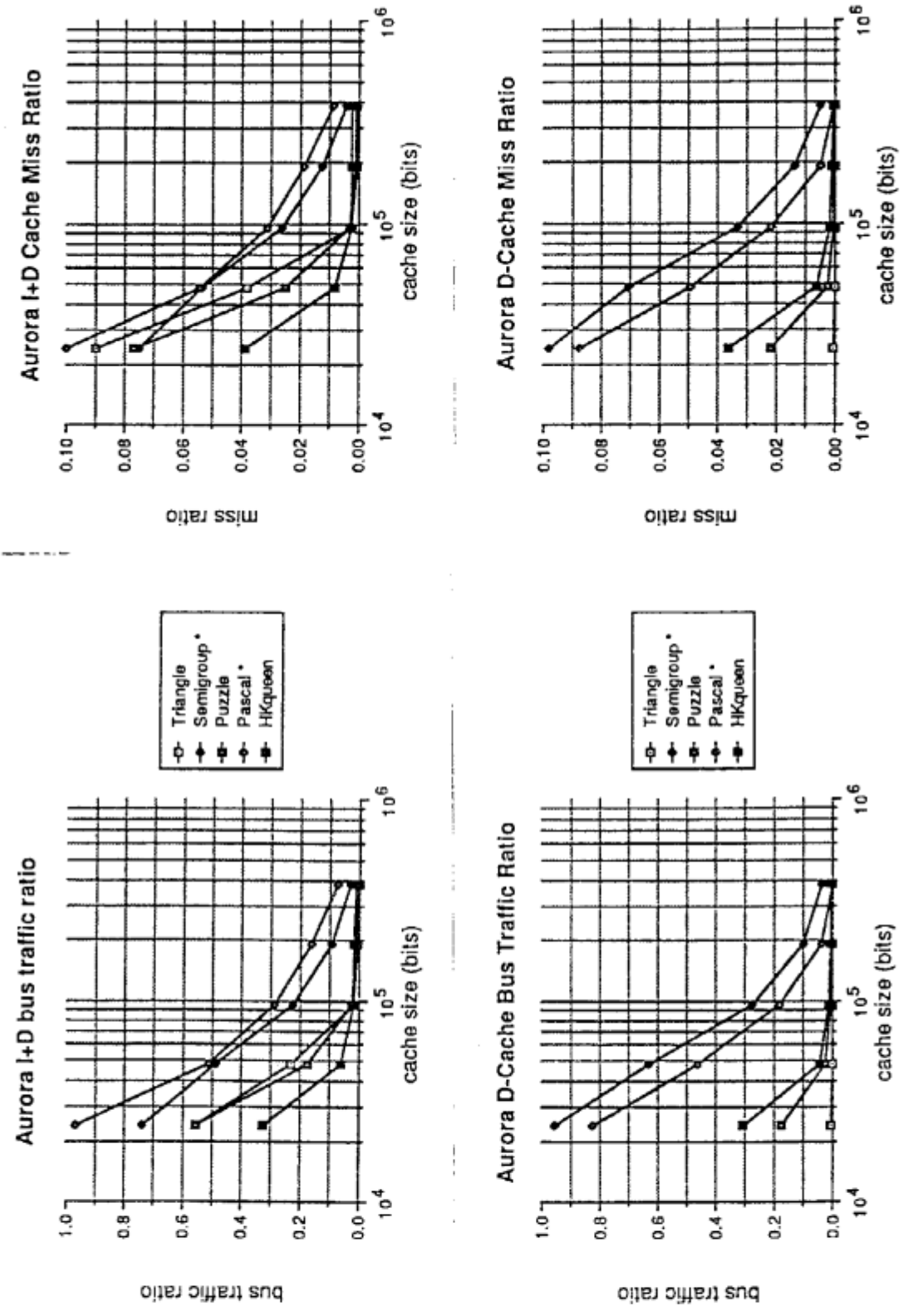


Figure 13: Aurora Cache Performance: Miss and Bus Traffic Ratios

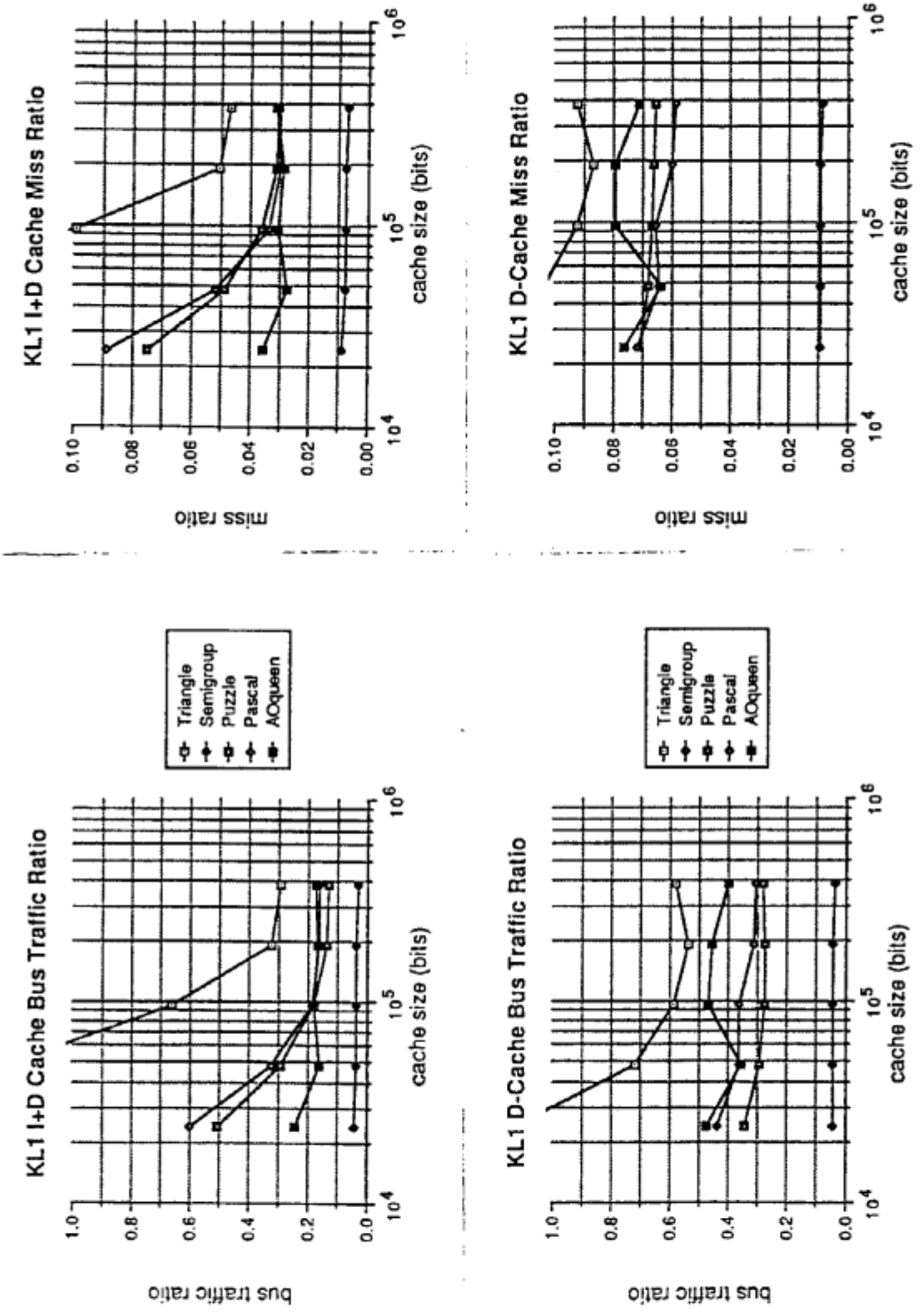


Figure 14: KL1 Cache Performance: Miss and Bus Traffic Ratios

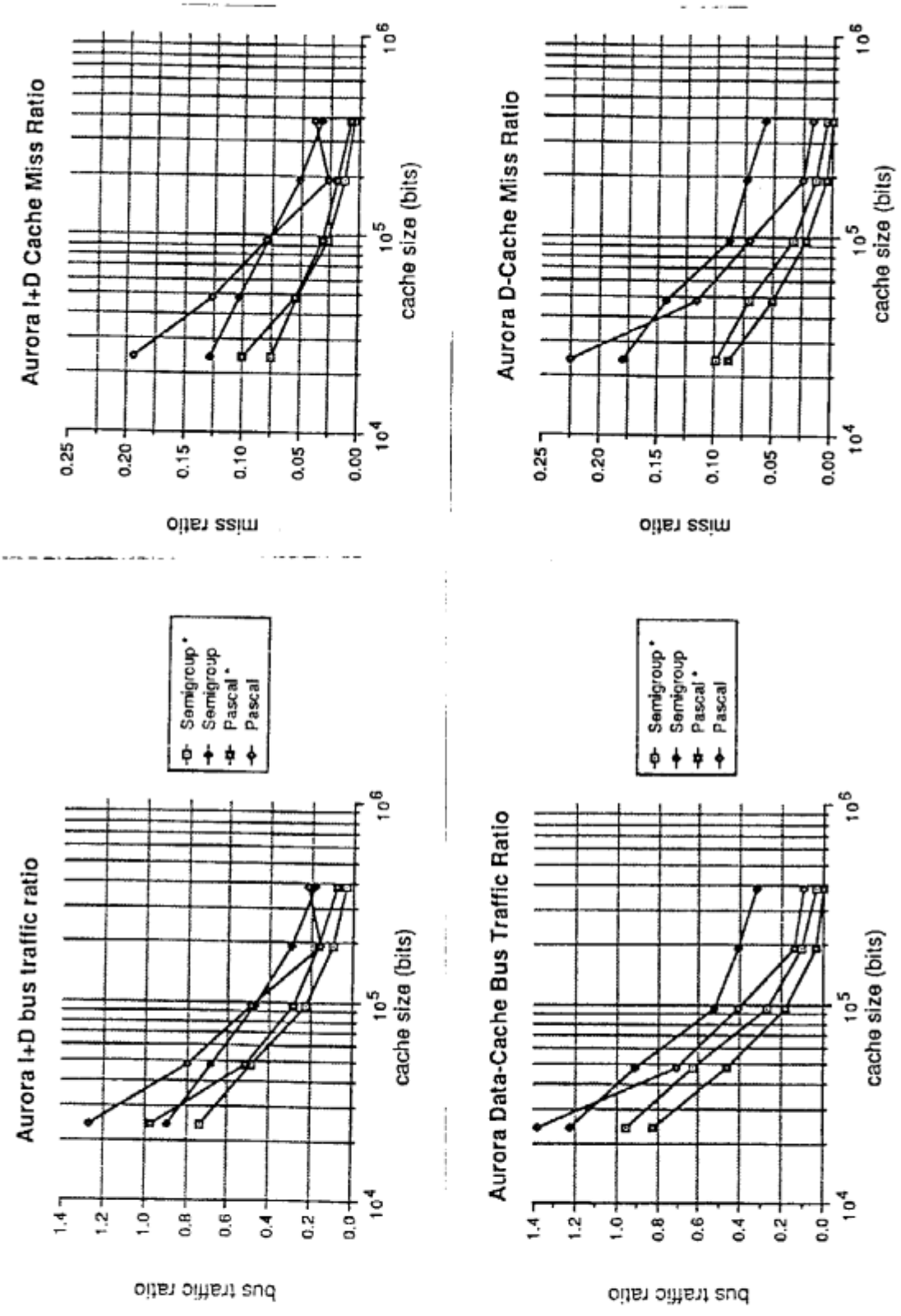


Figure 15: Aurora Scheduling Overheads: Two and Eight PE Comparison

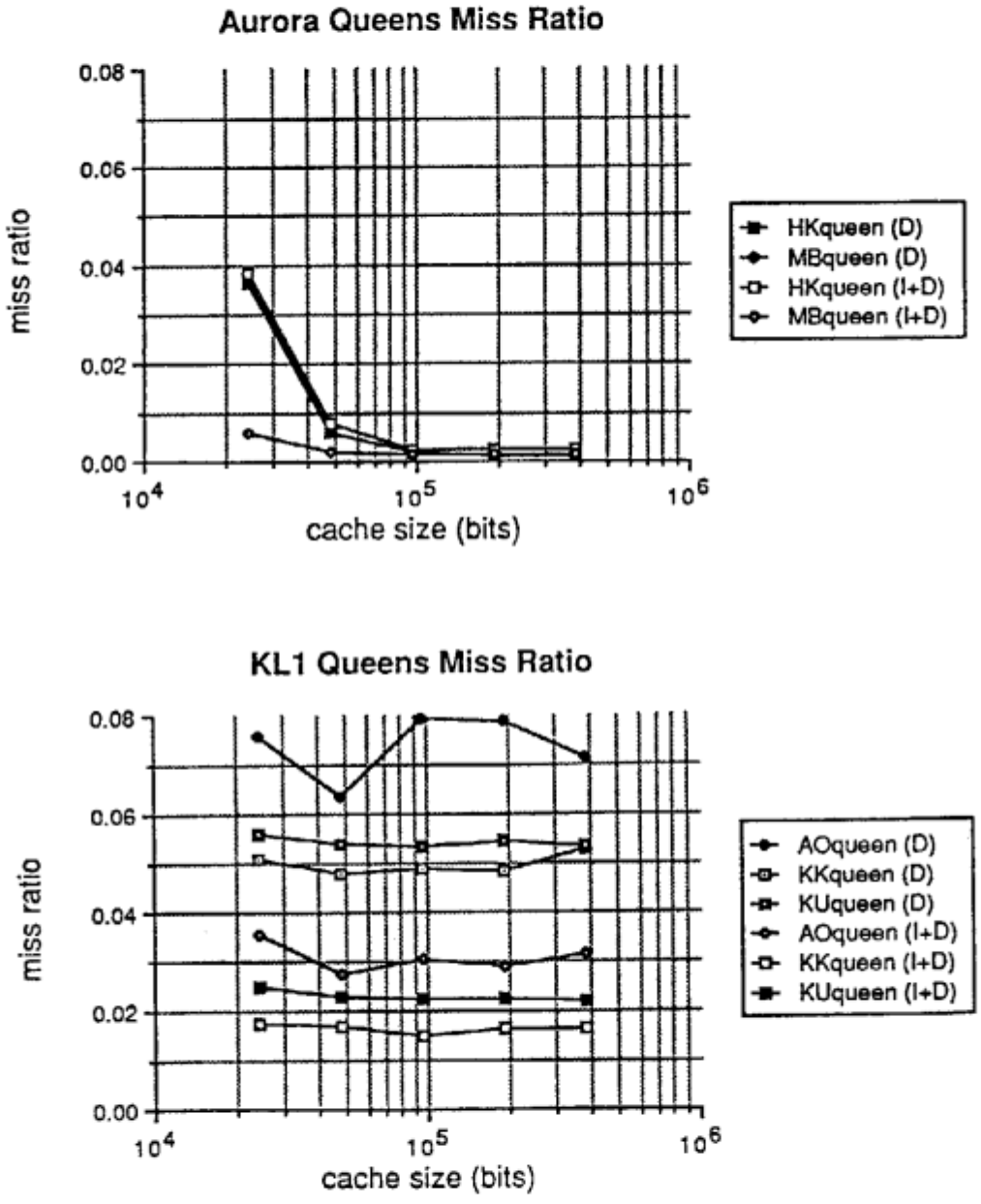


Figure 16: 10-Queens Comparison: Miss Ratio

architectures.

The most important result of this study was a confirmation that indeed *(independent) OR-parallel architectures have better memory performance than (dependent) AND-parallel architectures.* The reasons are that OR-parallel architectures can exploit an efficient stack-based storage model whereas dependent AND-parallel architectures must resort to a less efficient heap-based model. For all-solutions search problems, a further result is that *non-committed-choice architectures have better memory performance than committed-choice architectures.* This is because backtracking architectures can efficiently reclaim storage during all-solutions search, thereby reducing working-set size. Committed-choice architectures, like functional language architectures, consume memory at a rapid rate. Incremental GC can alleviate some of penalty for this memory appetite, but incremental GC also incurs its own overheads. Thirdly, for single-solution problems, *OR-parallel architectures cannot exploit parallelism as efficiently as dependent AND-parallel architectures can.* Although OR-parallel goals may exist, they are often too fine-grained for the excessive overheads necessary to execute them in parallel. In this respect, *dependent AND-parallel architectures can execute fine-grain parallelism more efficiently than can OR-parallel architectures.*

From the raw timings we saw that even with the anticipated 10% improvement in KL1 speed due to compiler optimization [43], 2–9 times improvement is needed to equal Aurora's speed. **Pascal** is the single exception where KL1 outperformed Aurora because the Argonne scheduler went crazy trying to find parallelism that did not exist. In any case, both systems calibrated on a simple determinate benchmark. There is no doubt however that the benchmarks favor Prolog. **Triangle** was translated from Prolog to FGHC, thereby incurring overheads. **Puzzle** in KL1 involves excessive structure copying. **Semigroup** in KL1 does not use a 2-3 tree as does Prolog. **AOqueens** uses layered-streams, thereby incurring suspensions. On the other hand, *is there a more natural or more efficient way to write these programs in FGHC?* Prolog-to-FGHC continuation-based translation, layered-streams, object-oriented programming, pipelined parallelism, etc. are all publicized methods of parallel FGHC programming. If there are better versions of these programs, it would be very enlightening to measure them.

A "bottom-heavy" system, such as KL1, makes a tradeoff between the ease of exploiting parallelism and the power of language constructs. Backtracking and full logical unification have been traded for stream-AND parallelism. There is a loose analogy in the tradeoff made by Prolog with respect to Lisp. Prolog makes a tradeoff between declarative semantics and the power of language constructs. We also see the about the same performance ratio between Prolog and Lisp [54] and Prolog and FGHC. On the other hand, Aurora makes a tradeoff between the power of language constructs and the availability of parallelism. This leads to dismal results for programs with no OR-parallelism like **Pascal** or the compiler studied by Carlsson [10]. There

is a large class of problems, not represented here, requiring intelligent search strategies, e.g., **Maxflow** and **Bestpath**. These problems cannot be solved efficiently by Aurora.

11 Future Work

The benchmarks in this study are too small and in the future should be replaced by more realistic application programs. Benchmark development is a troublesome problem in a young field such as this where language and architecture definitions are constantly changing, and where system implementations are immature. Additional algorithms should be developed for the problems already analyzed. For example, the FGHC version of **Semigroup** should be rewritten to use a data/process structure equivalent to the 2-3 tree.

The Aurora OR-parallel Prolog architecture exhibits data sharing characteristics that are highly centralized. All processes frequently access the same shared node tree (control stack). This study presents measurements of invalidation broadcast caches only; however, for Aurora, update broadcast appears to be more matched to centralized sharing. In the future an update broadcast protocol should be measured.

The KL1 AND-parallel FGHC architecture displays a high bandwidth requirement, similar to that of functional programming language architectures. Thus these architectures require garbage collection (GC) subsystems. In the KL1 system measured in this study, a naive stop-and-copy GC was implemented. This has the advantage of allowing large benchmarks to be tested, but because it is not incremental, it does not significantly reduce the bandwidth requirement. Incremental GC schemes such as MRB[12] provide this ability. Nishida[38] claims MRB can reduce KL1 bus traffic (on a shared memory multiprocessor model) by 15-26% on eight PEs. The measurements presented in this paper should be extended to include optional incremental GC for comparison.

Empirical studies such as this one make many assumptions and approximations to facilitate making measurements. The mapping of the emulator state onto the target architecture state is especially difficult and error prone. To obtain more accurate measurements, i.e., measurements that more closely model the real system that is being designed, this mapping must be made more exact. For example, in the Aurora and KL1 systems, various global data structures are used to represent information about each PE. In this study, references to such data structures are not regarded as abstract memory references. In more accurate models, perhaps these structures will be accessed from memory.

Read-purge and read-buffer cache operations should be instrumented in the KL1 real-parallel system. Comparison with Matsumoto's earlier results indicate that these operations may reduce bus traffic by more than expected.

12 Acknowledgements

This research was supported in full by NSF Grant No. IRI-8704576. The project was conducted at the Institute for New Generation Computer Technology (ICOT). The author thanks the Director of ICOT, Dr. Kazuhiro Fuchi, and Dr. Shun-ichi Uchida for supporting his stay at ICOT in terms of resources, work environment, and general day-to-day life.

I thank Professor Michael Flynn and Susan Gere of Stanford University for their understanding and support, difficult at best from such a great distance.

I thank my co-workers at ICOT for helping with this research. Most notable are M. Sato, who wrote the parallel KL1 emulator, and A. Matsumoto, who wrote the parallel cache simulator. They both helped me modify, debug and analyze these systems. I owe a great deal to A. Okumura who helped develop **Triangle**, **Pascal**, and many other KL1 benchmarks. I also had informative discussions with A. Goto, N. Ichiyoshi, Y. Kimura, K. Ueda, and others.

I thank S. Haridi of the Swedish Institute of Computer Science (SICS), and R. Lusk and R. Overbeek of Argonne National Laboratories (ANL) for supplying me with the Aurora OR-Parallel Prolog system. Discussions and assistance from R. Overbeek and R. Stevens from ANL were helpful in understanding this system. I also thank A. Ciepielewski of SICS whose help instrumenting and analyzing Aurora during his three month visit at ICOT was invaluable.

I thank I. Foster of Imperial College and M. Hermenegildo of Microelectronics Computer Technology Corporation (MCC) for informative discussions about "the big picture."

A Appendix: Prolog Benchmarks

A.1 Triangle

```
/*----------------------------------------------------------------------------
Program: Triangle (all solutions, OR-parallel)
Author:  E. Tick
Date:    August 7 1988
Notes:

1. To run:
       ?- go(T,N).
where output T is the execution time and output N should be 133.

2. The initial board:
                              1
                            1   1
                          1   0   1
                        1   1   1   1
                      1   1   1   1   1
   which is represented by the structure:
              b(1,1,1,1,0, 1,1,1,1,1, 1,1,1,1,1)

   is simplified by making the first three moves,
   to reduce the solution space:

                              1
                            1   1
                          1   1   1
                        1   0   1   1
                      1   0   1   1   1
   which is represented by the structure:
              b(1,1,1,1,1, 1,1,0,1,1, 1,0,1,1,1)

                              1
                            1   1
                          1   1   1
                        1   0   1   1
                      1   1   0   0   1
   which is represented by the structure:
              b(1,1,1,1,1, 1,1,0,1,1, 1,1,0,0,1)

                              1
                            1   1
                          1   1   0
                        1   0   0   1
                      1   1   1   0   1
   which is represented by the structure:
              b(1,1,1,1,1, 0,1,0,0,1, 1,1,1,0,1)

   The goal of this game is to remove all the pegs (1's) from
   the board.  Any peg can jump over any other peg
   along a straight line and land in an open hole.  The jumped
   peg is removed.  This translates into 36 possible moves.

   The goal of the triangle benchmark is to calculate all
   winning sequences of moves (there are 133 given these first
   three moves).  Winning sequences are collected with a bagof --
   the solutions are then counted.  The program can be greatly
   lengthened by removing the initial forced moves.
----------------------------------------------------------------------------*/
```

```
:- parallel move/3.

go(T,N) :-
    time(_),
    bagof(X,play(3,b(1,1,1,1,1, 0,1,0,0,1, 1,1,1,0,1),X),L),
    time(T),
    count(L,N).

time(T) :- statistics(runtime,[_,T]).

count(L,N) :- count(L,0,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).
count([],N,N).

play(13,_,[]) :- !.
play(M,Board,[P|X]) :-
    move(P,Board,NewBoard),
    M1 is M+1,
    play(M1,NewBoard,X).

move(1,b( 1, 1,X3, 0,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15),
    b( 0, 0,X3, 1,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
move(2,b(X1, 1,X3, 1,X5,X6, 0,X8,X9,X10,X11,X12,X13,X14,X15),
    b(X1, 0,X3, 0,X5,X6, 1,X8,X9,X10,X11,X12,X13,X14,X15)).
move(3,b(X1,X2,X3, 1,X5,X6, 1,X8,X9,X10,  0,X12,X13,X14,X15),
    b(X1,X2,X3, 0,X5,X6, 0,X8,X9,X10,  1,X12,X13,X14,X15)).
move(4,b(X1,X2, 1,X4, 1,X6,X7, 0,X9,X10,X11,X12,X13,X14,X15),
    b(X1,X2, 0,X4, 0,X6,X7, 1,X9,X10,X11,X12,X13,X14,X15)).
move(5,b(X1,X2,X3,X4, 1,X6,X7, 1,X9,X10,X11,  0,X13,X14,X15),
    b(X1,X2,X3,X4, 0,X6,X7, 0,X9,X10,X11,  1,X13,X14,X15)).
move(6,b(X1,X2,X3,X4,X5, 1,X7,X8, 1,X10,X11,X12,  0,X14,X15),
    b(X1,X2,X3,X4,X5, 0,X7,X8, 0,X10,X11,X12,  1,X14,X15)).
move(7,b( 1,X2, 1,X4,X5, 0,X7,X8,X9,X10,X11,X12,X13,X14,X15),
    b( 0,X2, 0,X4,X5, 1,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
move(8,b(X1,X2, 1,X4,X5, 1,X7,X8,X9,  0,X11,X12,X13,X14,X15),
    b(X1,X2, 0,X4,X5, 0,X7,X8,X9,  1,X11,X12,X13,X14,X15)).
move(9,b(X1,X2,X3,X4,X5, 1,X7,X8,X9,  1,X11,X12,X13,X14,  0),
    b(X1,X2,X3,X4,X5, 0,X7,X8,X9,  0,X11,X12,X13,X14,  1)).
move(10,b(X1, 1,X3,X4, 1,X6,X7,X8, 0,X10,X11,X12,X13,X14,X15),
     b(X1, 0,X3,X4, 0,X6,X7,X8, 1,X10,X11,X12,X13,X14,X15)).
move(11,b(X1,X2,X3,X4, 1,X6,X7,X8, 1,X10,X11,X12,X13,  0,X15),
     b(X1,X2,X3,X4, 0,X6,X7,X8, 0,X10,X11,X12,X13,  1,X15)).
move(12,b(X1,X2,X3, 1,X5,X6,X7, 1,X9,X10,X11,X12,  0,X14,X15),
     b(X1,X2,X3, 0,X5,X6,X7, 0,X9,X10,X11,X12,  1,X14,X15)).
move(13,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  1,  1,  0,X14,X15),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  0,  0,  1,X14,X15)).
move(14,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  1,  1,  0,X15),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  0,  0,  1,X15)).
move(15,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  1,  1,  0),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  0,  0,  1)).
move(16,b(X1,X2,X3,X4,X5,X6, 1, 1, 0,X10,X11,X12,X13,X14,X15),
     b(X1,X2,X3,X4,X5,X6, 0, 0, 1,X10,X11,X12,X13,X14,X15)).
move(17,b(X1,X2,X3,X4,X5,X6,X7, 1, 1,  0,X11,X12,X13,X14,X15),
     b(X1,X2,X3,X4,X5,X6,X7, 0, 0,  1,X11,X12,X13,X14,X15)).
move(18,b(X1,X2,X3, 1, 1, 0,X7,X8,X9,X10,X11,X12,X13,X14,X15),
     b(X1,X2,X3, 0, 0, 1,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
move(19,b( 0, 1,X3, 1,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15),
     b( 1, 0,X3, 0,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
move(20,b(X1, 0,X3, 1,X5,X6, 1,X8,X9,X10,X11,X12,X13,X14,X15),
     b(X1, 1,X3, 0,X5,X6, 0,X8,X9,X10,X11,X12,X13,X14,X15)).
move(21,b(X1,X2,X3, 0,X5,X6, 1,X8,X9,X10,  1,X12,X13,X14,X15),
     b(X1,X2,X3, 1,X5,X6, 0,X8,X9,X10,  0,X12,X13,X14,X15)).
```

```
move(22,b(X1,X2, 0,X4, 1,X6,X7, 1,X9,X10,X11,X12,X13,X14,X15),
     b(X1,X2, 1,X4, 0,X6,X7, 0,X9,X10,X11,X12,X13,X14,X15)).
move(23,b(X1,X2,X3,X4, 0,X6,X7, 1,X9,X10,X11,  1,X13,X14,X15),
     b(X1,X2,X3,X4, 1,X6,X7, 0,X9,X10,X11,  0,X13,X14,X15)).
move(24,b(X1,X2,X3,X4,X5, 0,X7,X8, 1,X10,X11,X12,  1,X14,X15),
     b(X1,X2,X3,X4,X5, 1,X7,X8, 0,X10,X11,X12,  0,X14,X15)).
move(25,b( 0,X2, 1,X4,X5, 1,X7,X8,X9,X10,X11,X12,X13,X14,X15),
     b( 1,X2, 0,X4,X5, 0,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
move(26,b(X1,X2, 0,X4,X5, 1,X7,X8,X9,  1,X11,X12,X13,X14,X15),
     b(X1,X2, 1,X4,X5, 0,X7,X8,X9,  0,X11,X12,X13,X14,X15)).
move(27,b(X1,X2,X3,X4,X5, 0,X7,X8,X9,  1,X11,X12,X13,X14,  1),
     b(X1,X2,X3,X4,X5, 1,X7,X8,X9,  0,X11,X12,X13,X14,  0)).
move(28,b(X1, 0,X3,X4, 1,X6,X7,X8, 1,X10,X11,X12,X13,X14,X15),
     b(X1, 1,X3,X4, 0,X6,X7,X8, 0,X10,X11,X12,X13,X14,X15)).
move(29,b(X1,X2,X3,X4, 0,X6,X7,X8, 1,X10,X11,X12,X13,  1,X15),
     b(X1,X2,X3,X4, 1,X6,X7,X8, 0,X10,X11,X12,X13,  0,X15)).
move(30,b(X1,X2,X3, 0,X5,X6,X7, 1,X9,X10,X11,X12,  1,X14,X15),
     b(X1,X2,X3, 1,X5,X6,X7, 0,X9,X10,X11,X12,  0,X14,X15)).
move(31,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  0,  1,  1,X14,X15),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,  1,  0,  0,X14,X15)).
move(32,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  0,  1,  1,X15),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,  1,  0,  0,X15)).
move(33,b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  0,  1,  1),
     b(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,  1,  0,  0)).
move(34,b(X1,X2,X3,X4,X5,X6, 0, 1, 1,X10,X11,X12,X13,X14,X15),
     b(X1,X2,X3,X4,X5,X6, 1, 0, 0,X10,X11,X12,X13,X14,X15)).
move(35,b(X1,X2,X3,X4,X5,X6,X7, 0, 1,  1,X11,X12,X13,X14,X15),
     b(X1,X2,X3,X4,X5,X6,X7, 1, 0,  0,X11,X12,X13,X14,X15)).
move(36,b(X1,X2,X3, 0, 1, 1,X7,X8,X9,X10,X11,X12,X13,X14,X15),
     b(X1,X2,X3, 1, 0, 0,X7,X8,X9,X10,X11,X12,X13,X14,X15)).
```

A.2 Puzzle

```
/*-------------------------------------------------------------------------
Program: Puzzle (all solutions, OR-parallel)
Author:  E. Tick
Date:    July 4 1988

Notes:
1. To run:
    ?- go(T,N).
where output T = time and N = 65 (number of solutions).

2. This is 5x4x3 puzzle with chip in corner.

3. This version collects answers in a list.  Each answer is a list of seven
pieces indicating the origin at which they were placed.
-------------------------------------------------------------------------*/
:- parallel fill/4,p321/3,p431/3,p331/3,p421/3.

go(T,N) :-
    time(_),
    make_board(Board),
    initialize(Board, Pieces),
    findall(Game, play(Board, Pieces, Game,0), L),
    count(L,N),
    time(T).

% chip-off corner to remove symmetry...
initialize([s(z,_,_,_)|_],[[a,b,c], [d,e], [f], [g]]).

play([],_,[],_).                          % game over
play([s(V,_,_,_)|Rest],Pieces,Ns,M) :-  % spot already filled
    nonvar(V),!,
    M1 is M+1,
    play(Rest,Pieces,Ns,M1).
play([Spot|Rest],Pieces,[[N|M]|Ns],M) :-
    fill(Spot,N,Pieces,NewPieces),      % spot empty - try to fill
    M1 is M+1,
    play(Rest,NewPieces,Ns,M1).

fill(Board,N,[[Mark|P1]|T],[P1|T])                  :- p321(Mark,N,Board).
fill(Board,N,[P1,[Mark|P2]|T],[P1,P2|T])            :- p431(Mark,N,Board).
fill(Board,N,[P1,P2,[Mark|P3]|T],[P1,P2,P3|T])      :- p331(Mark,N,Board).
fill(Board,N,[P1,P2,P3,[Mark|P4]|T],[P1,P2,P3,P4|T]) :- p421(Mark,N,Board).

% piece templates:

% p1 = 4x2x1: 4 orientations
p421(M, a, s(M,                                % 4-2-1
             s(M,
                s(M,
                   s(M,_,
                       s(M,_,_,_),
                       _),
                   s(M,_,_,_),
                   _),
                s(M,_,_,_),
                _),
             s(M,_,_,_),
             _)).

p421(M, b, s(M, _,                             % 1-4-2
             s(M,_,
```

```
                s(M,_,
                    s(M,_,
                        _,
                        s(M,_,_,_)),
                    s(M,_,_,_)),
                s(M,_,_,_)),
            s(M,_,_,_))).

p421(M, c, s(M,                                 % 2-4-1
            s(M,_,_,_),
            s(M,
                s(M,_,_,_),
                s(M,
                    s(M,_,_,_),
                    s(M,
                        s(M,_,_,_),
                        _,
                        _),
                    _),
                _),
            _)).


p421(M, d, s(M,                                 % 4-1-2
            s(M,
                s(M,
                    s(M,_,
                        _,
                        s(M,_,_,_)),
                    _,
                    s(M,_,_,_)),
                _,
                s(M,_,_,_)),
            _,
            s(M,_,_,_))).

% p331 = 3x3x1: 3 orientations
p331(M, e, s(M,                                 % 3-3-1
            s(M,
                s(M,_,
                    s(M,_,s(M,_,_,_),_),
                    _),
                s(M,_,s(M,_,_,_),_),
                _),
            s(M,_,s(M,_,_,_),_),
            _)).

p331(M, f, s(M,                                 % 3-1-3
            s(M,
                s(M,_,
                    _,
                    s(M,_,_,s(M,_,_,_))),
                _,
                s(M,_,_,s(M,_,_,_))),
            _,
            s(M,_,_,s(M,_,_,_)))).

p331(M, g, s(M,                                 % 1-3-3
            _,
            s(M,
                _,
                s(M,_,
```

```
                    _,
                    s(M,_,_,s(M,_,_,_))),
                s(M,_,_,s(M,_,_,_))),
            s(M,_,_,s(M,_,_,_)))).

% p321 = 3x2x1: 6 orientations
p321(M, h, s(M,                                    % 3-2-1
                    s(M,
                        s(M,_,
                            s(M,_,_,_),
                            _),
                        s(M,_,_,_),
                        _),
                    s(M,_,_,_),
                    _)).

p321(M, i, s(M,                                    % 2-1-3
                    s(M,_,_,_),
                    _,
                    s(M,
                        s(M,_,_,_),
                        _,
                        s(M,
                            s(M,_,_,_),
                            _,
                            _)))).

p321(M, j, s(M, _,                                 % 1-3-2
          s(M,_,
              s(M,_,
                  _,
                  s(M,_,_,_)),
              s(M,_,_,_)),
          s(M,_,_,_))).

p321(M, k, s(M,                                    % 2-3-1
             s(M,_,_,_),
             s(M,
                 s(M,_,_,_),
                 s(M,
                     s(M,_,_,_),
                     _,
                     _),
                 _),
             _)).

p321(M, l, s(M,                                    % 3-1-2
             s(M,
                 s(M,
                     _,
                     _,
                     s(M,_,_,_)),
                 _,
                 s(M,_,_,_)),
             _,
             s(M,_,_,_))).

p321(M, m, s(M, _,                                 % 1-2-3
             s(M,_,_,_),
             s(M,_,
                 s(M,_,_,_),
                 s(M,_,
```

```
                    s(M,_,_,_),
                    _)))).

% p431 = 4x3x1: 4 orientations
p431(M, n, s(M,                                    % 4-3-1
          s(M,
              s(M,
                  s(M,_,
                      s(M,_,s(M,_,_,_),_),
                      _),
                  s(M,_,s(M,_,_,_),_),
                  _),
              s(M,_,s(M,_,_,_),_),
              _),
          s(M,_,s(M,_,_,_),_),
          _)).

p431(M, o, s(M, _,                                 % 1-4-3
              s(M,_,
                  s(M,_,
                      s(M,_,
                          _,
                          s(M,_,_,s(M,_,_,_))),
                      s(M,_,_,s(M,_,_,_))),
                  s(M,_,_,s(M,_,_,_))),
              s(M,_,_,s(M,_,_,_)))).

p431(M, p, s(M,                                    % 3-4-1
              s(M,s(M,_,_,_),_,_),
              s(M,
                  s(M,s(M,_,_,_),_,_),
                  s(M,
                      s(M,s(M,_,_,_),_,_),
                      s(M,
                          s(M,s(M,_,_,_),_,_),
                          _,
                          _),
                      _),
                  _),
              _)).

p431(M, q, s(M,                                    % 4-1-3
              s(M,
                  s(M,
                      s(M,_,
                          _,
                          s(M,_,_,s(M,_,_,_))),
                      _,
                      s(M,_,_,s(M,_,_,_))),
                  _,
                  s(M,_,_,s(M,_,_,_))),
              _,
              s(M,_,_,s(M,_,_,_)))).

make_board(Level0) :-
    make_level(Level0-Level1,Level1-_),
    make_level(Level1-Level2,Level2-_),
    make_level(Level2-[],[z,z,z,z,z,z, z,z,z,z,z,z,
                          z,z,z,z,z,z, z,z,z,z,z]-[]).

make_level(C-Link,Z-L) :-
    C= [C00,C10,C20,C30,C40,
```

```
        C01,C11,C21,C31,C41,
        C02,C12,C22,C32,C42,
        C03,C13,C23,C33,C43|Link],

    Z= [Z00,Z10,Z20,Z30,Z40,
        Z01,Z11,Z21,Z31,Z41,
        Z02,Z12,Z22,Z32,Z42,
        Z03,Z13,Z23,Z33,Z43|L],

    node(C10,C01,Z00, N1, N2,C00),
    node(C20,C11,Z10, N2, N3,C10),
    node(C30,C21,Z20, N3, N4,C20),
    node(C40,C31,Z30, N4, N5,C30),
    node(  z,C41,Z40, N5, N6,C40),

    node(C11,C02,Z01, N6, N7,C01),
    node(C21,C12,Z11, N7, N8,C11),
    node(C31,C22,Z21, N8, N9,C21),
    node(C41,C32,Z31, N9,N10,C31),
    node(  z,C42,Z41,N10,N11,C41),

    node(C12,C03,Z02,N11,N12,C02),
    node(C22,C13,Z12,N12,N13,C12),
    node(C32,C23,Z22,N13,N14,C22),
    node(C42,C33,Z32,N14,N15,C32),
    node(  z,C43,Z42,N15,N16,C42),

    node(C13,  z,Z03,N16,N17,C03),
    node(C23,  z,Z13,N17,N18,C13),
    node(C33,  z,Z23,N18,N19,C23),
    node(C43,  z,Z33,N19,N20,C33),
    node(  z,  z,Z43,N20,  _,C43).

node(X,Y,Z,N,N,s(_,X,Y,Z)).

time(T) :- statistics(runtime,[_,T]).

count(L,N) :- count(L,0,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).
count([],N,N).
```

A.3 Pascal

```
/*-------------------------------------------------------------------------
Program:  Pascal's Triangle
Author:   E. Tick with BIGNUM package written by R. O'Keefe
Date:     August 17 1988

Notes:
1. To run:
    ?- go(N,R,T).
where N is the input row number and the output R is a list of coefficients
and T is the execution time.

2. example: for N = 20:
R = [[1,0],[20,0],[190,0],[1140,0],[4845,0],[15504,0],[38760,0],[77520,0],
     [25970,1],[67960,1],[84756,1],[67960,1],[25970,1],[77520,0],[38760,0],
     [15504,0],[4845,0],[1140,0],[190,0],[20,0],[1,0]]

3. This version uses assert/retract to perform its own GC because Aurora does
not support GC.  This is necessary because the algorithm uses up heap space
at a fast rate.

4. How this version works: there is a maximum granularity (chosen to be six).
The row of Pascal's triangle to be calculated is divided into chunks equal
to the maximum granularity.  Each chunk is spawned in AND-parallel with
the remaining portion of the row to be processed.  The end of the row is
sequentialized because GC must be implemented at the source-level with a
fail.  Main factors limiting execution speed:
        1. home-brew GC, implemented by a fail, is invoked for each
           row calculation, and therefore parallelism cannot overlap
           row calculations.

5. This version uses new AND-in-OR parallel scheme optimized for this
specific case of two-way determinate parallelism.  In this version,
the merge of solutions from the left and right children is done efficiently
without member.
-------------------------------------------------------------------------*/
:- parallel gather_sols/3.

go(N, R, T) :-
    time(_),
    N > 0,
    assert(row([[1,0],[1,0]])),                    % seed row
    pascal(1, N, R),
    time(T).

time(T) :- statistics(runtime,[_,T]).

pascal(N, N, R) :- !, retract(row(R)).            % clean up after...
pascal(K, N, R) :-
    make_pascal(K),
    K1 is K + 1,
    pascal(K1, N, R).

make_pascal(K) :-
    retract(row([F|Data])),
    W = 6,                                  % 6 is max granularity for now
    Odd is K mod 2,
    H is (K+1)//2,
    Iter is H // W,
    End is H mod W,
    make_row(Iter,End,[F|Data],Result,[F],Odd),
    assert(row([F|Result])),
```

```
    fail.                                   % homebrew GC
make_pascal(_).

make_row(0,End,In,Out,Rev,Odd) :- !,        % final padding
    granule(End,In,Out,Rev,Odd).
make_row(N,End,In,Out,Rev,Odd) :-           % straight-away
    N1 is N-1,
    big_granule(N1,End,In,Out,Rev,Odd).

finish_row(0, Rev,    Rev) :- !.            % even row end case
finish_row(1, Rev,[_|Rev]).                 %  odd row end case

granule(0,In,Out,Rev,Odd) :- !,
    finish_row(Odd, Out, Rev).
granule(1,[A,B|Rest],[AB|R],T,Odd) :- !,
    big_plus(AB,A,B),
    finish_row(Odd, R, [AB|T]).
granule(2,[A,B,C|Rest],[AB,BC|R],T,Odd) :- !,
    big_plus(AB,A,B),
    big_plus(BC,B,C),
    finish_row(Odd, R, [BC,AB|T]).
granule(3,[A,B,C,D|Rest],[AB,BC,CD|R],T,Odd) :- !,
    big_plus(AB,A,B),
    big_plus(BC,B,C),
    big_plus(CD,C,D),
    finish_row(Odd, R, [CD,BC,AB|T]).
granule(4,[A,B,C,D,E|Rest],[AB,BC,CD,DE|R],T,Odd) :- !,
    big_plus(AB,A,B),
    big_plus(BC,B,C),
    big_plus(CD,C,D),
    big_plus(DE,D,E),
    finish_row(Odd, R, [DE,CD,BC,AB|T]).
granule(5,[A,B,C,D,E,F|Rest],[AB,BC,CD,DE,EF|R],T,Odd) :-
    big_plus(AB,A,B),
    big_plus(BC,B,C),
    big_plus(CD,C,D),
    big_plus(DE,D,E),
    big_plus(EF,E,F),
    finish_row(Odd, R, [EF,DE,CD,BC,AB|T]).

big_granule(N,End,[A,B,C,D,E,F,G|Rest],[AB,BC,CD,DE,EF,FG|R],T,Odd) :-
    and(make_row(N,End,[G|Rest],R,[FG,EF,DE,CD,BC,AB|T],Odd),
        work(A,B,C,D,E,F,G,AB,BC,CD,DE,EF,FG)).

work(A,B,C,D,E,F,G,AB,BC,CD,DE,EF,FG) :-
        big_plus(AB,A,B),
        big_plus(BC,B,C),
        big_plus(CD,C,D),
        big_plus(DE,D,E),
        big_plus(EF,E,F),
        big_plus(FG,F,G).

/*-------------------------------------------------------------------------
Program:  AND-in-OR parallelism for two determinate goals
Author:   E. Tick (based on original from M. Carlsson)
Notes:
1. gets about same speedup as standard version, but is more efficient.
Speedup is a function of the granularity of the goals, but the overhead
of joining them determines the absolute speed.

2. schemes to improve efficiency of merge unification have failed.  It
appears that simply unifying Goal1 and Goal2 directly is most efficient,
```

```
even though these structures may be complex.
*/

and(Goal1,Goal2) :-
    findall(Sol, gather_sols(Sol,Goal1,Goal2), Sols),
    (Sols = [s1(Goal1),s2(Goal2)] ; Sols = [s2(Goal2),s1(Goal1)]),!.

gather_sols(s1(Goal1),Goal1,_):- call(Goal1).
gather_sols(s2(Goal2),_,Goal2):- call(Goal2).

/*-------------------------------------------------------------------------
Program: BIGNUM package
Author:  R. O'Keefe
*/

% this interface is meant to save storage...
big_plus(X,Y,Z) :- eval(real(+,X,[1]) is real(+,Y,[1]) + real(+,Z,[1])),!.
big_grt(X,Y) :- eval(real(+,X,[1]) > real(+,Y,[1])),!.

eval(compare(X,Y,S)) :- eval(X, A), eval(Y, B), comq(A, B, 100000, R), !, R=S.
eval(X < Y)          :- eval(compare(X,Y,S)), !, S=(<).
eval(X > Y)          :- eval(compare(X,Y,S)), !, S=(>).
eval(B is Y)         :- eval(Y, B).

eval(X+Y, C)     :- !, eval(X, A), eval(Y, B), addq(A, B, 100000, C).
eval(X,X).

%%% comq
comq(A,A,_,=) :- !.
comq(real(+,Na,Da), real(+,Nb,Db), R, S) :-
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),    !,
        comn(Xa, Xb, =, S).
comq(real(+,Na,Da), real(-,Nb,Db), R, >) :- !.
comq(real(-,Na,Da), real(+,Nb,Db), R, <) :- !.
comq(real(-,Na,Da), real(-,Nb,Db), R, S) :-
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),    !,
        comn(Xb, Xa, =, S).
comq(Na, real(+,Nb,Db), R, S) :- Na >= 0,
        muln([Na], Db, R, Xa),  !,
        comn(Xa, Nb, =, S).
comq(Na, real(-,Nb,Db), R, >) :- Na >= 0, !.
comq(Na, real(+,Nb,Db), R, <) :- !.
comq(Na, real(-,Nb,Db), R, S) :- Nz is - Na,
        muln([Nz], Db, R, Xa),
        comn(Nb, Xa, =, S).
comq(real(+,Na,Da), Nb, R, S) :- Nb >= 0,
        muln([Nb], Da, R, Xb),    !,
        comn(Na, Xb, =, S).
comq(real(+,Na,Da), Nb, R, >) :- !.
comq(real(-,Na,Da), Nb, R, <) :- Nb >= 0, !.
comq(real(-,Na,Da), Nb, R, S) :- Nz is - Nb,
        muln([Nz], Da, R, Xb),    !,
        comn(Xb, Na, =, S).
comq(Na, Nb, R, >) :- Na > Nb, !.
comq(Na, Nb, R, <) :- !.

%%% addq
addq(A, B, R, S) :-
        real(A, R, Sa, Na, Da),
        real(B, R, Sb, Nb, Db),
```

```
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),
        addz(Sa,Xa, Sb,Xb, R, Sc,Xc),
        gcdn(Xc, Da, R, _, Nx, Ya),
        gcdn(Nx, Db, R, _, Nc, Yb),
        muln(Ya, Yb, R, Dc), Nc/Dc\==[]/[],
        standardise(real(Sc, Nc, Dc),S), !.


        muln([], B, R, []) :- !.
        muln(A, [], R, []) :- !.
        muln(A,  B, R,  C) :- !, muln(A, B, [], R, C).

        muln([D1|T1], N2, Ac, R, [D3|Pr]) :-
                mul1(N2, D1, R, P2),
                addn(Ac, P2, 0, R, Sm),
                conn(D3, An, Sm), !,
                muln(T1, N2, An,R, Pr).
        muln([],      N2, Ac, R, Ac) :- !.

                mul1(A, 0, R, []) :- !.
                mul1(A, M, R, Pr) :- !,
                        mul1(A, M, 0, R, Pr).

                        mul1([], M, 0, R, []) :- !.
                        mul1([], M, C, R, [C]) :- !.
                        mul1([D1|T1], M, C, R, [D2|T2]) :-
                                D2 is (D1*M+C) mod R,
                                Co is (D1*M+C)  //  R,
                                mul1(T1, M, Co, R, T2).


%%% addz
addz(+,A, +,B, R, +,C) :- !, addn(A, B, 0, R, C).
addz(+,A, -,B, R, S,C) :- !, subn(A, B, R, S, C).
addz(-,A, +,B, R, S,C) :- !, subn(B, A, R, S, C).
addz(-,A, -,B, R, -,C) :- !, addn(B, A, 0, R, C).

        addn([D1|T1], [D2|T2], Cin, R, [D3|T3]) :-
                Sum is D1+D2+Cin,
                X is Sum mod 262144,
                (   X >= R, Cout = 1, D3 is X-R
                ;   X <  R, Cout = 0, D3 =  Sum
                ),  !,
                addn(T1, T2, Cout, R, T3).
        addn([], L, 0, R, L) :- !.
        addn([], L, 1, R, M) :- !, add1(L, R, M).
        addn(L, [], 0, R, L) :- !.
        addn(L, [], 1, R, M) :- !, add1(L, R, M).

                add1([M|T], R, [N|T]) :- N is M+1, N < R, !.
                add1([M|T], R, [0|S]) :- R is M+1, !, add1(T, R, S).
                add1([],    R, [1]).


%%% gcdn
        gcdn([], [], R,  [], undefined, undefined) :- !.
        gcdn([],  B, R,   B,  [], [1]) :- !.
        gcdn( A, [], R,   A, [1],  []) :- !.
        gcdn([1], B, R, [1], [1],   B) :- !.    %  common case
        gcdn( A,[1], R, [1],   A, [1]) :- !.    %  common case
        gcdn( A,  B, R,   D,   M,   N) :-       %  A, B > 1
```

```
        gcdn(A, B, R, D),
        divn(A, D, R, M, _),
        divn(B, D, R, N, _).

        gcdn(A, B, R, D) :-              %  A, B >= 1  !!
                comn(A, B, =, S), !,
                gcdn(S, A, B, R, D).

                gcdn(<,[], B, R, B) :- !.
                gcdn(<, A, B, R, D) :-
                        estg(B, A, R, E),
                        muln(E, A, R, P),
                        subn(B, P, R, _, M), !,
                        gcdn(A, M, R, D).
                gcdn(>, A,[], R, A) :- !.
                gcdn(>, A, B, R, D) :-
                        estg(A, B, R, E),
                        muln(E, B, R, P),
                        subn(A, P, R, _, M), !,
                        gcdn(M, B, R, D).
                gcdn(=, A, B, R, A).

                        estg(    A,   [B], R, E) :- !,
                                div1(A, B, R, Q, X),
                                (   X*2 =< B, E = Q
                                ;   add1(Q, R, E)
                                ), !.
                        estg([_|A], [_|B], R, E) :- !,
                                estg(A, B, R, E).
```

%%% divn

```
        divn(A, [], R, _, _) :- !, fail. % division by 0 is undefined
        divn(A,[1], R, A,[]) :- !.       % a very common special case
        divn(A,[B], R, Q, X) :- !,       % nearly as common a case
                div1(A, B, R, Q, Y),
                conn(Y, [], X).
        divn(A,  B, R, Q, X) :-
                comn(A, B, =, S),
                (   S = '<', Q =  [], X = A
                ;   S = '=', Q = [1], X = []
                ), !.
        divn(A,  B, R, Q, X) :- !,
                divm(A, B, R, Q, X).

                conn(0, [],   []) :- !.
                conn(D, T, [D|T]).

                div1([D1|T1], B1, R, Q1, X1) :- !,
                        div1(T1, B1, R, Q2, X2),
                        D2 is (X2*R+D1)  //  B1,
                        X1 is (X2*R+D1) mod B1,
                        conn(D2, Q2, Q1).
                div1([],      B1, R, [],  0).

% divm(A, B, R, Q, X) is called with A > B > R

                divm([D1|T1], B, R, Q1, X1) :- !,
                        divm(T1, B, R, Q2, X2),
                        conn(D1, X2, T2),
                        div2(T2, B, R, D2, X1),
                        conn(D2, Q2, Q1).
```

```
divm([],      B, R, [], []).

        div2(A, B, R, Q, X) :-
                estd(A, B, R, E), !,
                chkd(A, B, R, E, 0, Q, P), !,
                subn(A, P, R, S, X).   %  S=+
        div2(A, B, R, _, _) .  %   long_error(divq, A/B).

                estd([A0,A1,A2], [B0,B1], R, E) :-
                        B1 >= R/2, !,
                        E is (A2*R+A1)/B1.
                estd([A0,A1,A2], [B0,B1], R, E) :- !,
                        L is (A2*R+A1)/(B1+1),
                        mul1([B0,B1],    L, R, P),
                        subn([A0,A1,A2], P, R, S, N), !, %S=+
                        estd(N, [B0,B1], R, M),    !,
                        E is L+M.
                estd([A0,A1],    [B0,B1], R, E) :- !,
                        E is (A1*R+A0+1)/(B1*R+B0).
                estd([A0],       _,       R, 0) :- !.
                estd([A0|Ar],    [B0|Br], R, E) :- !,
                        estd(Ar, Br, R, E).
                estd([],         _,       R, 0) :- !.

                chkd(A, B, R, E, 3, _, _) :- !.
                        %        long_error(divq, A/B).
                chkd(A, B, R, E, K, E, P) :-
                        mul1(B, E, R, P),
                        comn(P, A, <, <), !.
                chkd(A, B, R, E, K, Q, P) :-
                        L is K+1, F is E-1, !,
                        chkd(A, B, R, F, L, Q, P).

%%% subn
        subn(A, B, R, S, C) :-
                comn(A, B, =, 0), !,  %  Oh for Ordering
                subn(0, A, B, R, S, C).

                subn(<, A, B, R, -, C) :- !, subp(B, A, 0, R, D), prune(D, C).
                subn(>, A, B, R, +, C) :- !, subp(A, B, 0, R, D), prune(D, C).
                subn(=, A, B, R, +,[]) :- !.

                        prune([0|L], M ) :- !,
                                prune(L, T),
                                (T = [], M = []; M = [0|T]).
                        prune([D|L], [D|M]) :- !,
                                prune(L, M).
                        prune([],    []) :- !.

                subp([D1|T1], [D2|T2], Bin, R, [D3|T3]) :-
                        S is D1-D2-Bin,
                        (   S >= 0, Bout = 0, D3 =  S
                        ;   S <  0, Bout = 1, D3 is S+R
                        ),  !,
                        subp(T1, T2, Bout, R, T3).
                subp(L, [], 0, R, L) :- !.
                subp(L, [], 1, R, M) :- !, sub1(L, R, M).

                        sub1([0|T], R, [K|S]) :- !, K is R-1, sub1(T, R, S).
                        sub1([N|T], R, [M|T]) :- M is N-1.
```

```
%%%  comn

        comn([D1|T1], [D2|T2], D, S) :-
                com1(D1, D2, D, N), !,
                comn(T1, T2, N, S).
        comn([],      [],      D, S) :- !, S = D.
        comn([],      L,       D, <) :- !.
        comn(L,       [],      D, >) :- !.

                com1(X, X, D, D) :- !.
                com1(X, Y, D, <) :- X < Y, !.
                com1(X, Y, D, >) :- X > Y, !.

%%% real/4
real(undefined, R, +, [], []) :- !.
real(real(S, N, D), R, S,  N,  D) :- !.
real(N, R, +, L, [1]) :- integer(N), N >= 0, !,          binrad(N, R, L).
real(N, R, -, L, [1]) :- integer(N), N  < 0, !, M is -N, binrad(M, R, L).

        binrad(0, R, [])    :- !.
        binrad(N, R, [M|T]) :- K is N//R, M is N mod R, !, binrad(K, R, T).

%%% standardise
standardise(real(S,[N],[1]), Ans) :- !,
        (   S = '+', Ans = N
        ;   S = '-', Ans is -N
        ),  !.
standardise(real(S, N, []), undefined) :- !.
standardise(real(_, [],[1]),  0 ) :- !.
standardise(Number,         Number).
```

A.4 Semigroup

```
/*-------------------------------------------------------------------------
Program:  Semigroup (all-solutions OR-Parallel)
Author:   R. Overbeek
Modified: E. Tick
Date:     August 20 1988

Notes:
1. To run:
    ?- go(T,N).
where T is time and N should be output 313.

2. this version is reputed to be fastest so far, but still uses 2-3 trees.

3. this version includes the generators in the answer (KL1 version doesn't)

4. this version has tuple length hardwired: BE CAREFUL!

5. this version is NON-DETERMINANT: it gets very slightly different numbers
of reductions and instructions executed on Aurora for 1--8 PEs!  I don't know
the reason for this...

6. This program gets poor speedup because the granularity of the parallelism
(a findall of newtup/4) is limited.
-------------------------------------------------------------------------*/
:- parallel member/2, umember/2.

go(T,N) :-
    init_sos(Sos,Sub),
    time(_),
    gen_products(Sos,Sub,Hbg,Sos),
    time(T),
    count(Hbg, N).

init_sos(Sos,Sub) :-
    sos(Sos),
    extend_tree(Sos,nil,Sub).

% state(Sos, Sub, Hbg)
%       Sos = list of tuples that need to be processed
%       Sub = tree corresponding to these tuples
%       Hbg = semigroup tuples (initially [])
%
gen_products(Sos,Sub,Hbg,Kernel) :-
    gen_all(state(Sos,Sub,[]),state(_,_,Hbg),Kernel).

gen_all(state([],Sub,Hbg),state([],Sub,Hbg),_) :- !.
gen_all(S, F, Kernel) :-
    gen_one(S, S1, Kernel),
    gen_all(S1, F, Kernel).

gen_one(state([H,I|T], Sub,  Hbg),
        state( Sos1, Sub1, [H,I|Hbg]), Kernel) :- !,
    findall(Tuple, newtup([H,I], Kernel, Sub, Tuple), L),
    proc_new(L, Sub, Sub1, T, Sos1).
gen_one(state([H|T], Sub,  Hbg),
        state( Sos1, Sub1, [H|Hbg]), Kernel) :-
    findall(Tuple, newtup([H], Kernel, Sub, Tuple), L),
    proc_new(L, Sub, Sub1, T, Sos1).

% proc_new(L, Sub, Sub1, Sos, Sos1):
%       L = list of candidate tuples to be possibly added to queue
```

```
%       Sub = tree describing current queue
%       Sub1 = new tree after all L tuples have been processed
%       Sos = current queue
%       Sos1 = new queue after all L tuples have been processed
% if L is empty, then tree and queue remain the same...
proc_new(   [], Sub,  Sub, Sos,  Sos).
% process non-empty L: declaratively, if processing T = tail(L) results
% in new tree Sub2 and new queue Sos2, then we consider two cases of trying
% to add first tuple H to Sub2:
%       if H can be added (doesn't exist already), then
%             Sub1 = new tree
%             Sos1 = new queue
%       otherwise (H cannot be added because it exists already), then
%             Sub1 = Sub2
%             Sos1 = Sos2
proc_new([H|T], Sub, Sub1, Sos, Sos1) :-
    proc_new(T, Sub, Sub2, Sos, Sos2),
    (add23(Sub2,H,Sub1) ->
        Sos1 = [H|Sos2]
        ;
        (Sub1 = Sub2, Sos1 = Sos2)).

newtup(E,L,Sub,New) :-
    member(E1,E),
    umember(E2,L),
    paired(E1,E2,New,Sub).

paired(E1,E2,New,Sub) :-
    bigm(E2,E1,New),
    \+ acc23(New,Sub).

bigm(W1,W2,P) :-
    functor(P,tuple,40),
    mtab(Table),
    bigm(1,W1,W2,P,Table).

bigm(41,_,_,_,_) :- !.
bigm(I,W0,W1,P,Table) :-  I < 41,
    arg(I,W0,X),
    arg(I,W1,Y),
    m(X,Y,Z,Table),
    arg(I,P,Z),
    J is (I + 1),
    bigm(J,W0,W1,P,Table).

m(X,Y,Z,M) :- arg(X,M,Row), arg(Y,Row,Z).

mtab(table(row(1,1,1,1,1),
           row(1,2,1,4,1),
           row(1,1,3,1,5),
           row(1,1,4,1,2),
           row(1,5,1,3,1))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% utilities...
member(H,[H|_]).
member(H,[_|T]) :- member(H,T).

umember(H,[H|_]).
umember(H,[_,H|T]).
umember(H,[_,_,H|T]).
umember(H,[_,_,_,H|T]).
```

```
umember(H,[_,_,_,_|T]) :- umember(H,T).

extend_tree([],S,S).
extend_tree([E|T],S,S1) :-
    add23(S,E,S2),
    extend_tree(T,S2,S1).

count(L,N) :- count(L,0,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).
count([],N,N).

time(T) :- statistics(runtime,[_,T]).

% 2-3 Trees: code from I. Bratko, "Prolog Programming for AI"
acc23(X,1(X)).
acc23(X,n2(T1,M,_)) :- M @> X, !, acc23(X,T1).
acc23(X,n2(_,_,T2)) :- acc23(X,T2).
acc23(X,n3(T1,M2,_,_,_)) :- M2 @> X, !, acc23(X,T1).
acc23(X,n3(_,_,T2,M3,_)) :- M3 @> X, !, acc23(X,T2).
acc23(X,n3(_,_,_,_,T3)) :- acc23(X,T3).

add23(Tree,X,Tree1) :-
    ins(Tree,X,Tree1).
add23(Tree,X,n2(T1,M2,T2)) :-
    ins(Tree,X,T1,M2,T2).

ins(nil,X,1(X)).
ins(n2(T1,M,T2),X,n2(NT1,M,T2)) :-
    M @> X,
    ins(T1,X,NT1).
ins(n2(T1,M,T2),X,n3(NT1a,Mb,NT1b,M,T2)) :-
    M @> X,
    ins(T1,X,NT1a,Mb,NT1b).
ins(n2(T1,M,T2),X,n2(T1,M,NT2)) :-
    X @> M,
    ins(T2,X,NT2).
ins(n2(T1,M,T2),X,n3(T1,M,NT2a,Mb,NT2b)) :-
    X @> M,
    ins(T2,X,NT2a,Mb,NT2b).
ins(n3(T1,M2,T2,M3,T3),X,n3(NT1,M2,T2,M3,T3)) :-
    M2 @> X,
    ins(T1,X,NT1).
ins(n3(T1,M2,T2,M3,T3),X,n3(T1,M2,NT2,M3,T3)) :-
    X @> M2,
    M3 @> X,
    ins(T2,X,NT2).
ins(n3(T1,M2,T2,M3,T3),X,n3(T1,M2,T2,M3,NT3)) :-
    X @> M3,
    ins(T3,X,NT3).

ins(1(A),X,1(A),X,1(X)) :-
    X @> A.
ins(1(A),X,1(X),A,1(A)) :-
    A @> X.
ins(n3(T1,M2,T2,M3,T3),X,n2(NT1a,Mb,NT1b),M2,n2(T2,M3,T3)) :-
    M2 @> X,
    ins(T1,X,NT1a,Mb,NT1b).
ins(n3(T1,M2,T2,M3,T3),X,n2(T1,M2,NT2a),Mb,n2(NT2b,M3,T3)) :-
    X @> M2,
    M3 @> X,
    ins(T2,X,NT2a,Mb,NT2b).
ins(n3(T1,M2,T2,M3,T3),X,n2(T1,M2,T2),M3,n2(NT3a,Mb,NT3b)) :-
```

```
    X @> M3,
    ins(T3,X,NT3a,Mb,NT3b).

% 309+4 solutions:
sos([tuple(1,1,1,1,1, 2,2,2,2,2, 3,3,3,3,3, 4,4,4,4,4, 5,5,5,5,5,
                                 3,3,3,3,3, 5,5,5,5,5, 4,4,4,4,4),
     tuple(1,2,3,4,5, 1,2,3,4,5, 1,2,3,4,5, 1,2,3,4,5, 1,2,3,4,5,
                                 1,2,3,4,5, 1,3,2,4,5, 1,2,3,4,5),
     tuple(1,1,1,1,1, 2,2,2,2,2, 3,3,3,3,3, 5,5,5,5,5, 4,4,4,4,4,
                      2,2,2,2,2,             4,4,4,4,4, 3,3,3,3,3),
     tuple(1,2,3,5,4, 1,2,3,5,4, 1,2,3,5,4, 1,2,3,5,4, 1,2,3,5,4,
                                 1,2,3,4,5, 1,2,3,5,4, 1,2,3,5,4)]).
```

A.5 Queens

A.5.1 HKqueen

```
/*------------------------------------------------------------------------
Program: 10-Queens (all-solutions, OR-parallel)
Author:  H. Kondo
Date:    May 18 1988

Notes:
1. To run:
    ?- go(N,T).
where output N is 724 (number of solutions) and T is execution time.
------------------------------------------------------------------------*/
:- parallel gen/3.

go(N, T) :-
    time(_),
    bagof(Q, P^(pattern(P),main(P, [1,2,3,4,5,6,7,8,9,10], [], Q)), S),
    count(S,N),
    time(T).

gen(0, []) :- !.
gen(N, [N|X]) :- M is N-1, gen(M,X).

count(L,N) :- count(L,0,N).
count([],N,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).

time(T) :- statistics(runtime,[_,T]).

main([H|T],L,Y,Z):- gen(L1,E,L), arg(E,H,a(E,E)), main(T,L1,[E|Y],Z).
main([],   _,Y,Y).

gen(L,    E,[E|L]).
gen([F|L],E,[F|P]):- gen(L,E,P).

pattern([                    % 10-Queens board...
b(a(Xa,Yj),a(Xb,Yi),a(Xc,Yh),a(Xd,Yg),a(Xe,Yf),
  a(Xf,Ye),a(Xg,Yd),a(Xh,Yc),a(Xi,Yb),a(Xj,Ya)),
b(a(X9,Yi),a(Xa,Yh),a(Xb,Yg),a(Xc,Yf),a(Xd,Ye),
  a(Xe,Yd),a(Xf,Yc),a(Xg,Yb),a(Xh,Ya),a(Xi,Y9)),
b(a(X8,Yh),a(X9,Yg),a(Xa,Yf),a(Xb,Ye),a(Xc,Yd),
  a(Xd,Yc),a(Xe,Yb),a(Xf,Ya),a(Xg,Y9),a(Xh,Y8)),
b(a(X7,Yg),a(X8,Yf),a(X9,Ye),a(Xa,Yd),a(Xb,Yc),
  a(Xc,Yb),a(Xd,Ya),a(Xe,Y9),a(Xf,Y8),a(Xg,Y7)),
b(a(X6,Yf),a(X7,Ye),a(X8,Yd),a(X9,Yc),a(Xa,Yb),
  a(Xb,Ya),a(Xc,Y9),a(Xd,Y8),a(Xe,Y7),a(Xf,Y6)),
b(a(X5,Ye),a(X6,Yd),a(X7,Yc),a(X8,Yb),a(X9,Ya),
  a(Xa,Y9),a(Xb,Y8),a(Xc,Y7),a(Xd,Y6),a(Xe,Y5)),
b(a(X4,Yd),a(X5,Yc),a(X6,Yb),a(X7,Ya),a(X8,Y9),
  a(X9,Y8),a(Xa,Y7),a(Xb,Y6),a(Xc,Y5),a(Xd,Y4)),
b(a(X3,Yc),a(X4,Yb),a(X5,Ya),a(X6,Y9),a(X7,Y8),
  a(X8,Y7),a(X9,Y6),a(Xa,Y5),a(Xb,Y4),a(Xc,Y3)),
b(a(X2,Yb),a(X3,Ya),a(X4,Y9),a(X5,Y8),a(X6,Y7),
  a(X7,Y6),a(X8,Y5),a(X9,Y4),a(Xa,Y3),a(Xb,Y2)),
b(a(X1,Ya),a(X2,Y9),a(X3,Y8),a(X4,Y7),a(X5,Y6),
  a(X6,Y5),a(X7,Y4),a(X8,Y3),a(X9,Y2),a(Xa,Y1))]).
```

A.5.2 MBqueen

```
/*-------------------------------------------------------------------------
Program: N-Queens (all solutions, OR-parallel)
Author:  M. Bruynooghe
Date:    June 14 1988

Notes:
1. To run:
    ?- go(M,N,T).
for example, for input M=8, output N=92 (number of solutions) and T is
execution time.
-------------------------------------------------------------------------*/
:- parallel del/3.

go(M,N,T) :- gen(M,L), time(_), bagof(X,queen(L,[],X),A), time(T), count(A,N).

queen([],R,P) :- rev(R,[],P).
queen([H|T], R, P) :- del([H|T],A,L),safe(R,A,1),queen(L,[A|R],P).

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

del([X|T], X, T).
del([H|T], X, [H|R]) :- del(T, X, R).

safe([],_,_).
safe([H|T],U,N) :- H+N=\=U, H-N=\=U, M is N+1, safe(T,U,M).

time(T) :- statistics(runtime,[_,T]).

count(L,N) :- count(L,0,N).
count([],N,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).

gen(0, []) :- !.
gen(N, [N|X]) :- M is N-1, gen(M,X).
```

A.5.3 IBqueen

```
/*-------------------------------------------------------------------------
Program: N-Queens (all-solutions, OR-parallel)
Author:  I. Bratko
Date:    June 14 1988

Notes:
1. To run:
    :- go(M,N,T).
for example, when input M=8, should return output N=92 (number of solutions)
and T is execution time.
-------------------------------------------------------------------------*/
:- parallel del/3.

go(M,N,T) :- time(_), bagof(X, queen(M,X), A), time(T), count(A,N).

queen(N,S) :-
    gen(1, N, Dxy),
    Nu1 is 1-N, Nu2 is N-1,
    gen(Nu1, Nu2, Du),
    Nv2 is N+N,
    gen(2, Nv2, Dv),
    sol(S, Dxy, Dxy, Du, Dv).

time(T) :- statistics(runtime,[_,T]).

count(L,N) :- count(L,0,N).
count([],    N,N).
count([X|Xs],M,N) :- M1 is M+1, count(Xs,M1,N).

sol([],[],Dy,Du,Dv).
sol([Y|Ylist],[X|Dx1],Dy,Du,Dv) :-
    del(Dy,Y,Dy1),
    U is X-Y,
    sdel(Du,U,Du1),
    V is X+Y,
    sdel(Dv,V,Dv1),
    sol(Ylist,Dx1,Dy1,Du1,Dv1).

% identical to del/3, but SEQUENTIAL
sdel([X|T], X, T).
sdel([H|T], X, [H|R]) :- sdel(T, X, R).

del([X|T], X, T).
del([H|T], X, [H|R]) :- del(T, X, R).

gen(N,N,[N]).
gen(N1,N2,[N1|L]) :- N1 < N2, M is N1+1, gen(M,N2,L).
```

B Appendix: FGHC Benchmarks

B.1 Triangle

```
/*------------------------------------------------------------------------
Program: Triangle (all-solutions AND-parallel)
Author:  A. Okumura (after E. Tick's Prolog version)
Date:    August 8 1988

Notes:
1. To run:
      ?- go(N).
output N should be 133.

2. This program has been automatically translated from Prolog, and then
optimized by hand using unfolding rules.
------------------------------------------------------------------------*/
go(N) :- true |
   'sweeper$playS'('L1'('L0'),3,b(1,1,1,1,1, 0,1,0,0,1, 1,1,1,0,1),A,[]),
    count(A,N).

count(L,N) :- true | count(L,0,N).
count([],M,N) :- true | M = N.
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).

'sweeper$playS'(A,B,C,D,E) :- true |
   'playS/3#1'(A,B,C,D,F), 'playS/3#2'(A,B,C,F,E).

'playS/3#1'(A,B,C,D,E) :- B<13 |
   'sweeper$smove'('L2'(A,B),C,D,E).
otherwise.
'playS/3#1'(A,B,C,D,E) :- true | D=E.

'playS/3#2'(A,13,B,C,D) :- true |
   'cont$playS/3'(A,[],C,D).
otherwise.
'playS/3#2'(A,B,C,D,E) :- true | D=E.

'cont$playS/3'('L3'(A,B),C,D,E) :- true |
   'cont$playS/3'(A,[B|C],D,E).
'cont$playS/3'('L1'('L0'),B,C,D) :- true | C = [B|D].

'sweeper$smove'(A,B,C,D) :- true |
    'smove/3#1'(A,B,C,E),  'smove/3#2'(A,B,E,F),  'smove/3#3'(A,B,F,G),
    'smove/3#4'(A,B,G,H),  'smove/3#5'(A,B,H,I),  'smove/3#6'(A,B,I,J),
    'smove/3#7'(A,B,J,K),  'smove/3#8'(A,B,K,L),  'smove/3#9'(A,B,L,M),
    'smove/3#10'(A,B,M,N), 'smove/3#11'(A,B,N,O), 'smove/3#12'(A,B,O,P),
    'smove/3#13'(A,B,P,Q), 'smove/3#14'(A,B,Q,R), 'smove/3#15'(A,B,R,S),
    'smove/3#16'(A,B,S,T), 'smove/3#17'(A,B,T,U), 'smove/3#18'(A,B,U,V),
    foobar(A,B,V,D).

foobar(A,B,V,D) :- true |
    'smove/3#19'(A,B,V,W),  'smove/3#20'(A,B,W,X),  'smove/3#21'(A,B,X,Y),
    'smove/3#22'(A,B,Y,Z),  'smove/3#23'(A,B,Z,A1), 'smove/3#24'(A,B,A1,B1),
    'smove/3#25'(A,B,B1,C1),'smove/3#26'(A,B,C1,D1),'smove/3#27'(A,B,D1,E1),
    'smove/3#28'(A,B,E1,F1),'smove/3#29'(A,B,F1,G1),'smove/3#30'(A,B,G1,H1),
    'smove/3#31'(A,B,H1,I1),'smove/3#32'(A,B,I1,J1),'smove/3#33'(A,B,J1,K1),
    'smove/3#34'(A,B,K1,L1),'smove/3#35'(A,B,L1,M1),'smove/3#36'(A,B,M1,D).

'smove/3#1'(A,b(1,1,B,0,C,D,E,F,G,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,1,b(0,0,B,1,C,D,E,F,G,H,I,J,K,L,M),N,O).
otherwise.
```

```
'smove/3#1'(A,BB,N,O) :- true | N=0.

'smove/3#2'(A,b(B,1,C,1,D,E,0,F,G,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,2,b(B,0,C,0,D,E,1,F,G,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#2'(A,BB,N,O) :- true | N=0.

'smove/3#3'(A,b(B,C,D,1,E,F,1,G,H,I,0,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,3,b(B,C,D,0,E,F,0,G,H,I,1,J,K,L,M),N,O).
otherwise.
'smove/3#3'(A,BB,N,O) :- true | N=0.

'smove/3#4'(A,b(B,C,1,D,1,E,F,0,G,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,4,b(B,C,0,D,0,E,F,1,G,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#4'(A,BB,N,O) :- true | N=0.

'smove/3#5'(A,b(B,C,D,E,1,F,G,1,H,I,J,0,K,L,M),N,O) :- true |
   'cont$smove/3'(A,5,b(B,C,D,E,0,F,G,0,H,I,J,1,K,L,M),N,O).
otherwise.
'smove/3#5'(A,BB,N,O) :- true | N=0.

'smove/3#6'(A,b(B,C,D,E,F,1,G,H,1,I,J,K,0,L,M),N,O) :- true |
   'cont$smove/3'(A,6,b(B,C,D,E,F,0,G,H,0,I,J,K,1,L,M),N,O).
otherwise.
'smove/3#6'(A,BB,N,O) :- true | N=0.

'smove/3#7'(A,b(1,B,1,C,D,0,E,F,G,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,7,b(0,B,0,C,D,1,E,F,G,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#7'(A,BB,N,O) :- true | N=0.

'smove/3#8'(A,b(B,C,1,D,E,1,F,G,H,0,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,8,b(B,C,0,D,E,0,F,G,H,1,I,J,K,L,M),N,O).
otherwise.
'smove/3#8'(A,BB,N,O) :- true | N=0.

'smove/3#9'(A,b(B,C,D,E,F,1,G,H,I,1,J,K,L,M,0),N,O) :- true |
   'cont$smove/3'(A,9,b(B,C,D,E,F,0,G,H,I,0,J,K,L,M,1),N,O).
otherwise.
'smove/3#9'(A,BB,N,O) :- true | N=0.

'smove/3#10'(A,b(B,1,C,D,1,E,F,G,0,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,10,b(B,0,C,D,0,E,F,G,1,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#10'(A,BB,N,O) :- true | N=0.

'smove/3#11'(A,b(B,C,D,E,1,F,G,H,1,I,J,K,L,0,M),N,O) :- true |
   'cont$smove/3'(A,11,b(B,C,D,E,0,F,G,H,0,I,J,K,L,1,M),N,O).
otherwise.
'smove/3#11'(A,BB,N,O) :- true | N=0.

'smove/3#12'(A,b(B,C,D,1,E,F,G,1,H,I,J,K,0,L,M),N,O) :- true |
   'cont$smove/3'(A,12,b(B,C,D,0,E,F,G,0,H,I,J,K,1,L,M),N,O).
otherwise.
'smove/3#12'(A,BB,N,O) :- true | N=0.

'smove/3#13'(A,b(B,C,D,E,F,G,H,I,J,K,1,1,0,L,M),N,O) :- true |
   'cont$smove/3'(A,13,b(B,C,D,E,F,G,H,I,J,K,0,0,1,L,M),N,O).
otherwise.
'smove/3#13'(A,BB,N,O) :- true | N=0.
```

```
'smove/3#14'(A,b(B,C,D,E,F,G,H,I,J,K,L,1,1,0,M),N,0) :- true |
   'cont$smove/3'(A,14,b(B,C,D,E,F,G,H,I,J,K,L,0,0,1,M),N,0).
otherwise.
'smove/3#14'(A,BB,N,0) :- true | N=0.

'smove/3#15'(A,b(B,C,D,E,F,G,H,I,J,K,L,M,1,1,0),N,0) :- true |
   'cont$smove/3'(A,15,b(B,C,D,E,F,G,H,I,J,K,L,M,0,0,1),N,0).
otherwise.
'smove/3#15'(A,BB,N,0) :- true | N=0.

'smove/3#16'(A,b(B,C,D,E,F,G,1,1,0,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,16,b(B,C,D,E,F,G,0,0,1,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#16'(A,BB,N,0) :- true | N=0.

'smove/3#17'(A,b(B,C,D,E,F,G,H,1,1,0,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,17,b(B,C,D,E,F,G,H,0,0,1,I,J,K,L,M),N,0).
otherwise.
'smove/3#17'(A,BB,N,0) :- true | N=0.

'smove/3#18'(A,b(B,C,D,1,1,0,E,F,G,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,18,b(B,C,D,0,0,1,E,F,G,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#18'(A,BB,N,0) :- true | N=0.

'smove/3#19'(A,b(0,1,B,1,C,D,E,F,G,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,19,b(1,0,B,0,C,D,E,F,G,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#19'(A,BB,N,0) :- true | N=0.

'smove/3#20'(A,b(B,0,C,1,D,E,1,F,G,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,20,b(B,1,C,0,D,E,0,F,G,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#20'(A,BB,N,0) :- true | N=0.

'smove/3#21'(A,b(B,C,D,0,E,F,1,G,H,I,1,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,21,b(B,C,D,1,E,F,0,G,H,I,0,J,K,L,M),N,0).
otherwise.
'smove/3#21'(A,BB,N,0) :- true | N=0.

'smove/3#22'(A,b(B,C,0,D,1,E,F,1,G,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,22,b(B,C,1,D,0,E,F,0,G,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#22'(A,BB,N,0) :- true | N=0.

'smove/3#23'(A,b(B,C,D,E,0,F,G,1,H,I,J,1,K,L,M),N,0) :- true |
   'cont$smove/3'(A,23,b(B,C,D,E,1,F,G,0,H,I,J,0,K,L,M),N,0).
otherwise.
'smove/3#23'(A,BB,N,0) :- true | N=0.

'smove/3#24'(A,b(B,C,D,E,F,0,G,H,1,I,J,K,1,L,M),N,0) :- true |
   'cont$smove/3'(A,24,b(B,C,D,E,F,1,G,H,0,I,J,K,0,L,M),N,0).
otherwise.
'smove/3#24'(A,BB,N,0) :- true | N=0.

'smove/3#25'(A,b(0,B,1,C,D,1,E,F,G,H,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,25,b(1,B,0,C,D,0,E,F,G,H,I,J,K,L,M),N,0).
otherwise.
'smove/3#25'(A,BB,N,0) :- true | N=0.

'smove/3#26'(A,b(B,C,0,D,E,1,F,G,H,1,I,J,K,L,M),N,0) :- true |
   'cont$smove/3'(A,26,b(B,C,1,D,E,0,F,G,H,0,I,J,K,L,M),N,0).
```

```
otherwise.
'smove/3#26'(A,BB,N,O) :- true | N=0.

'smove/3#27'(A,b(B,C,D,E,F,O,G,H,I,1,J,K,L,M,1),N,O) :- true |
   'cont$smove/3'(A,27,b(B,C,D,E,F,1,G,H,I,O,J,K,L,M,O),N,O).
otherwise.
'smove/3#27'(A,BB,N,O) :- true | N=0.

'smove/3#28'(A,b(B,O,C,D,1,E,F,G,1,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,28,b(B,1,C,D,O,E,F,G,O,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#28'(A,BB,N,O) :- true | N=0.

'smove/3#29'(A,b(B,C,D,E,O,F,G,H,1,I,J,K,L,1,M),N,O) :- true |
   'cont$smove/3'(A,29,b(B,C,D,E,1,F,G,H,O,I,J,K,L,O,M),N,O).
otherwise.
'smove/3#29'(A,BB,N,O) :- true | N=0.

'smove/3#30'(A,b(B,C,D,O,E,F,G,1,H,I,J,K,1,L,M),N,O) :- true |
   'cont$smove/3'(A,30,b(B,C,D,1,E,F,G,O,H,I,J,K,O,L,M),N,O).
otherwise.
'smove/3#30'(A,BB,N,O) :- true | N=0.

'smove/3#31'(A,b(B,C,D,E,F,G,H,I,J,K,O,1,1,L,M),N,O) :- true |
   'cont$smove/3'(A,31,b(B,C,D,E,F,G,H,I,J,K,1,O,O,L,M),N,O).
otherwise.
'smove/3#31'(A,BB,N,O) :- true | N=0.

'smove/3#32'(A,b(B,C,D,E,F,G,H,I,J,K,L,O,1,1,M),N,O) :- true |
   'cont$smove/3'(A,32,b(B,C,D,E,F,G,H,I,J,K,L,1,O,O,M),N,O).
otherwise.
'smove/3#32'(A,BB,N,O) :- true | N=0.

'smove/3#33'(A,b(B,C,D,E,F,G,H,I,J,K,L,M,O,1,1),N,O) :- true |
   'cont$smove/3'(A,33,b(B,C,D,E,F,G,H,I,J,K,L,M,1,O,O),N,O).
otherwise.
'smove/3#33'(A,BB,N,O) :- true | N=0.

'smove/3#34'(A,b(B,C,D,E,F,G,O,1,1,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,34,b(B,C,D,E,F,G,1,O,O,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#34'(A,BB,N,O) :- true | N=0.

'smove/3#35'(A,b(B,C,D,E,F,G,H,O,1,1,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,35,b(B,C,D,E,F,G,H,1,O,O,I,J,K,L,M),N,O).
otherwise.
'smove/3#35'(A,BB,N,O) :- true | N=0.

'smove/3#36'(A,b(B,C,D,O,1,1,E,F,G,H,I,J,K,L,M),N,O) :- true |
   'cont$smove/3'(A,36,b(B,C,D,1,O,O,E,F,G,H,I,J,K,L,M),N,O).
otherwise.
'smove/3#36'(A,BB,N,O) :- true | N=0.

'cont$smove/3'('L2'(A,B),C,D,E,F) :- G := B+1 |
   'sweeper$playS'('L3'(A,C),G,D,E,F).
```

B.2 Puzzle

```
/*-------------------------------------------------------------------------
Program: Puzzle (all solutions, AND-parallel)
Author:  E. Tick
Date:    March 9 1988

Notes:
1. To run:
    ?- go(N).
where output N = 65 (number of solutions).

2. This is 5x4x3 puzzle with chip in corner.  The program collects all
solutions in the form of a list of lists.  A solution list contains SEVEN
cons-cells corresponding to the pieces:
   [[b|26],[q|20],[j|16],[j|11],[l|5],[f|4],[a|1]]
The car represents the shape and orientation.  The cdr represents the
location it was placed inside the solid.
-------------------------------------------------------------------------*/
go(N) :- true |
    initial(Slist,Plist),
    select(Plist, Slist, [], A,[],. []),
    count(A, N).

% in this case, choose last instance of this shape...
select([orient(M,L)|Ys], Empty, NonC, I,0, PL):- M=:=1 |
    append(Ys, NonC, Unused),
    check(L, Unused, Empty,                I,I1, PL),
    select(Ys, Empty, [orient(M,L)|NonC], I1,0, PL).
% more than one instance of this shape exists...
select([orient(M,L)|Ys], Empty, NonC, I,0, PL):- M=\=1, M1 := M-1 |
    append([orient(M1,L)|Ys], NonC, Unused),
    check(L, Unused, Empty,                I,I1, PL),
    select(Ys, Empty, [orient(M,L)|NonC], I1,0, PL).
select([], _, [_|_], I,0,  _) :- true | I=0.
select([], _, [],    I,0, PL) :- true | I=[PL|0].

% The check routine is split into three parts for readability.
% Note however that this split does NOT slow it down: I ran a
% fused version with approximately the same execution speed (2% faster).
% This fact seems to imply that the speed-bump in this program is remove/5.

% spawn checker process for each orientation in Piece
check([D|Ds], Unused, Empty, I,0, PL) :- true |
    Empty = [E|RestEmpty],
    translate(D, E, Piece, Status),
    check1(Status, Ds, Piece, Unused, Empty, I,0, PL, [D|E], RestEmpty).
check([], _, _, I,0, _):- true | I=0.

% translated piece falls outside of solid boundary...
check1(no, Os, _, Unused, Empty, I,0, PL, _, _) :- true |
    check(  Os,    Unused, Empty, I,0, PL).
% translated piece falls completely inside of solid...
check1(yes, Os, Piece, Unused, Empty, I,0, PL, Move, RestEmpty) :- true |
    remove(yes, Piece, RestEmpty, NewEmpty, Status),
    check2(Status, Os, Unused, Empty, NewEmpty, I,0, PL, Move).

% translated piece falls inside a previously chosen piece...
check2(no, Os, Unused, Empty, _, I,0, PL, _) :- true |
    check(  Os, Unused, Empty,    I,0, PL).
% translated piece falls outside all previously chosen pieces...
check2(yes, Os, Unused, Empty, NewEmpty, I0,I2, PL, [D|o(X,Y,Z)]) :-
    M := X+(Y*5)+(Z*20) |        % calculate index of piece for answer...
```

```
    select(      Unused, NewEmpty, [],    I0,I1, [[D|M]|PL]),
    check(   Os, Unused, Empty,           I1,I2, PL).

% remove(yes, Vector, Empty, NewEmpty, Status)
% remove all elements in Vector from Empty
% return Status of removal:
%        "yes" if Vector was a subset of Empty
%        "no" if Vector contained elements not in Empty
remove( no,  _, _,  _, Status) :- true | Status = no.
remove(yes, [], Empty, T1, Status) :- true | Status = yes, T1 = Empty.
remove(yes, [H|T], Empty, T1, Status) :- true |
    remove2(Empty, H, T1, T2, NextEmpty, SubStatus),
    remove(SubStatus, T, NextEmpty, T2, Status).

remove2([], _, _, _, _, Status) :- true | Status = no.
remove2([E|Es], H, T1, T2, Empty, Status) :- E=H |
    Status = yes,
    Empty = Es,
    T1 = T2.
otherwise.
remove2([E|Es], H, T1, T3, Empty, Status) :- true |
    T1 = [E|T2],
    remove2(Es, H, T2, T3, Empty, Status).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 3x2x1 (6 orientations)
translate(a, o(X,Y,Z), List, Status) :- X<3, Y<3,
    X1 := X+1, X2 := X+2, Y1 := Y+1 | Status = yes,
    List = [          o(X1,Y, Z),o(X2,Y, Z),
            o(X,Y1,Z),o(X1,Y1,Z),o(X2,Y1,Z)].
translate(b, o(X,Y,Z), List, Status) :- X<3, Z<2,
    X1 := X+1, X2 := X+2, Z1 := Z+1 | Status = yes,
    List = [          o(X1,Y,Z), o(X2,Y,Z),
            o(X,Y,Z1),o(X1,Y,Z1),o(X2,Y,Z1)].
translate(c, o(X,Y,Z), List, Status) :- Y<2, Z<2,
    Y1 := Y+1, Y2 := Y+2, Z1 := Z+1 | Status = yes,
    List = [          o(X,Y1,Z), o(X,Y2,Z),
            o(X,Y,Z1),o(X,Y1,Z1),o(X,Y2,Z1)].
translate(d, o(X,Y,Z), List, Status) :- X<4, Y<2,
    Y1 := Y+1, Y2 := Y+2, X1 := X+1 | Status = yes,
    List = [          o(X, Y1,Z),o(X, Y2,Z),
            o(X1,Y,Z),o(X1,Y1,Z),o(X1,Y2,Z)].
translate(e, o(X,Y,Z), List, Status) :- X<4, Z<1,
    Z1 := Z+1, Z2 := Z+2, X1 := X+1 | Status = yes,
    List = [          o(X, Y,Z1),o(X, Y,Z2),
            o(X1,Y,Z),o(X1,Y,Z1),o(X1,Y,Z2)].
translate(f, o(X,Y,Z), List, Status) :- Y<3, Z<1,
    Z1 := Z+1, Z2 := Z+2, Y1 := Y+1 | Status = yes,
    List = [          o(X,Y, Z1),o(X,Y, Z2),
            o(X,Y1,Z),o(X,Y1,Z1),o(X,Y1,Z2)].

% 4x3x1 (4 orientations)
translate(g, o(X,Y,Z), List, Status) :- X<2, Y<2,
    X1 := X+1, X2 := X+2, X3 := X+3, Y1 := Y+1, Y2 := Y+2 | Status = yes,
    List = [          o(X1,Y, Z),o(X2,Y, Z),o(X3,Y, Z),
            o(X,Y1,Z),o(X1,Y1,Z),o(X2,Y1,Z),o(X3,Y1,Z),
            o(X,Y2,Z),o(X1,Y2,Z),o(X2,Y2,Z),o(X3,Y2,Z)].
translate(h, o(X,Y,Z), List, Status) :- Y<1, Z<1,
    Y1 := Y+1, Y2 := Y+2, Y3 := Y+3, Z1 := Z+1, Z2 := Z+2 | Status = yes,
    List = [          o(X,Y1,Z), o(X,Y2,Z), o(X,Y3,Z),
            o(X,Y,Z1),o(X,Y1,Z1),o(X,Y2,Z1),o(X,Y3,Z1),
```

```
            o(X,Y,Z2),o(X,Y1,Z2),o(X,Y2,Z2),o(X,Y3,Z2)].
translate(i, o(X,Y,Z), List, Status) :- X<3, Y<1,
    Y1 := Y+1, Y2 := Y+2, Y3 := Y+3, X1 := X+1, X2 := X+2 | Status = yes,
    List = [          o(X, Y1,Z),o(X, Y2,Z),o(X, Y3,Z),
            o(X1,Y,Z),o(X1,Y1,Z),o(X1,Y2,Z),o(X1,Y3,Z),
            o(X2,Y,Z),o(X2,Y1,Z),o(X2,Y2,Z),o(X2,Y3,Z)].
translate(j, o(X,Y,Z), List, Status) :- X<2, Z<1,
    X1 := X+1, X2 := X+2, X3 := X+3, Z1 := Z+1, Z2 := Z+2 | Status = yes,
    List = [          o(X1,Y,Z), o(X2,Y,Z), o(X3,Y,Z),
            o(X,Y,Z1),o(X1,Y,Z1),o(X2,Y,Z1),o(X3,Y,Z1),
            o(X,Y,Z2),o(X1,Y,Z2),o(X2,Y,Z2),o(X3,Y,Z2)].

% 3x3x1 (3 orientations)
translate(k, o(X,Y,Z), List, Status) :- X<3, Y<2,
    X1 := X+1, X2 := X+2, Y1 := Y+1, Y2 := Y+2 | Status = yes,
    List = [          o(X1,Y, Z),o(X2,Y, Z),
            o(X,Y1,Z),o(X1,Y1,Z),o(X2,Y1,Z),
            o(X,Y2,Z),o(X1,Y2,Z),o(X2,Y2,Z)].
translate(l, o(X,Y,Z), List, Status) :- Y<2, Z<1,
    Y1 := Y+1, Y2 := Y+2, Z1 := Z+1, Z2 := Z+2 | Status = yes,
    List = [          o(X,Y1,Z), o(X,Y2,Z),
            o(X,Y,Z1),o(X,Y1,Z1),o(X,Y2,Z1),
            o(X,Y,Z2),o(X,Y1,Z2),o(X,Y2,Z2)].
translate(m, o(X,Y,Z), List, Status) :- X<3, Z<1,
    X1 := X+1, X2 := X+2, Z1 := Z+1, Z2 := Z+2 | Status = yes,
    List = [          o(X1,Y,Z), o(X2,Y,Z),
            o(X,Y,Z1),o(X1,Y,Z1),o(X2,Y,Z1),
            o(X,Y,Z2),o(X1,Y,Z2),o(X2,Y,Z2)].

% 4x2x1 (4 orientations)
translate(n, o(X,Y,Z), List, Status) :- X<2, Y<3,
    X1 := X+1, X2 := X+2, X3 := X+3, Y1 := Y+1 | Status = yes,
    List = [          o(X1,Y, Z),o(X2,Y, Z),o(X3,Y, Z),
            o(X,Y1,Z),o(X1,Y1,Z),o(X2,Y1,Z),o(X3,Y1,Z)].
translate(o, o(X,Y,Z), List, Status) :- Y<1, Z<2,
    Y1 := Y+1, Y2 := Y+2, Y3 := Y+3, Z1 := Z+1 | Status = yes,
    List = [          o(X,Y1,Z), o(X,Y2,Z), o(X,Y3,Z),
            o(X,Y,Z1),o(X,Y1,Z1),o(X,Y2,Z1),o(X,Y3,Z1)].
translate(p, o(X,Y,Z), List, Status) :- Y<1, X<4,
    Y1 := Y+1, Y2 := Y+2, Y3 := Y+3, X1 := X+1 | Status = yes,
    List = [          o(X, Y1,Z),o(X, Y2,Z),o(X, Y3,Z),
            o(X1,Y,Z),o(X1,Y1,Z),o(X1,Y2,Z),o(X1,Y3,Z)].
translate(q, o(X,Y,Z), List, Status) :- X<2, Z<2,
    X1 := X+1, X2 := X+2, X3 := X+3, Z1 := Z+1 | Status = yes,
    List = [          o(X1,Y,Z), o(X2,Y,Z), o(X3,Y,Z),
            o(X,Y,Z1),o(X1,Y,Z1),o(X2,Y,Z1),o(X3,Y,Z1)].
otherwise.
translate(_, _, _, Status) :- true | Status = no.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% utilities...

append([A|X],Y,Z):- true | Z=[A|Z1], append(X,Y,Z1).
append([],   Y,Z):- true | Z=Y.

count(L,N) :- true | count(L,0,N).
count([],M,N) :- true | M = N.
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).

initial(Slist,Plist) :- true | squares(Slist), piece_list(Plist).
```

```
% NOTE: 4x3x1 and 4x2x1 have four orientations because z-dim. of puzzle
% is only three, so that these shapes cannot stand up in the z-direction.
piece_list(List) :- true |                  % shape  #  orientation
    List = [orient( 3,[a,b,c,d,e,f]),       % 3x2x1 (3)x(6)
            orient( 2,[g,h,i,j]),           % 4x3x1 (2)x(4)
            orient( 1,[k,l,m]),             % 3x3x1 (1)x(3)
            orient( 1,[n,o,p,q])].          % 4x2x1 (1)x(4)

squares(List) :- true |
    List =
    [         o(1,0,0),o(2,0,0),o(3,0,0),o(4,0,0),
     o(0,1,0),o(1,1,0),o(2,1,0),o(3,1,0),o(4,1,0),
     o(0,2,0),o(1,2,0),o(2,2,0),o(3,2,0),o(4,2,0),
     o(0,3,0),o(1,3,0),o(2,3,0),o(3,3,0),o(4,3,0),

     o(0,0,1),o(1,0,1),o(2,0,1),o(3,0,1),o(4,0,1),
     o(0,1,1),o(1,1,1),o(2,1,1),o(3,1,1),o(4,1,1),
     o(0,2,1),o(1,2,1),o(2,2,1),o(3,2,1),o(4,2,1),
     o(0,3,1),o(1,3,1),o(2,3,1),o(3,3,1),o(4,3,1),

     o(0,0,2),o(1,0,2),o(2,0,2),o(3,0,2),o(4,0,2),
     o(0,1,2),o(1,1,2),o(2,1,2),o(3,1,2),o(4,1,2),
     o(0,2,2),o(1,2,2),o(2,2,2),o(3,2,2),o(4,2,2),
     o(0,3,2),o(1,3,2),o(2,3,2),o(3,3,2),o(4,3,2)].
```

B.3 Pascal

```
/*-------------------------------------------------------------------------
Program:  Pascal's Triangle
Author:   E. Sugino
Modified: E. Tick
Date:     July 27 1988

Notes:
1. To run:
    ?- go(N,R).
where input N is the number of rows to calculate and output R is the Nth row.

2. example: for N = 20:
R = [[1,0],[20,0],[190,0],[1140,0],[4845,0],[15504,0],[38760,0],[77520,0],
     [25970,1],[67960,1],[84756,1],[67960,1],[25970,1],[77520,0],[38760,0],
     [15504,0],[4845,0],[1140,0],[190,0],[20,0],[1,0]]

2. This is a much simplified version of the original program, and only
calculates the Nth row from scratch.  Note that without bignums, we can
calculate as large as the 33rd row, with a maximum coefficient of
1,166,803,110.  This is equivalent to [3110,11668].
-------------------------------------------------------------------------*/
go(N) :- N > 0 |
    pascal_data([data(N,_)],1,[[1,0],[1,0]],1).

go(N, Result) :- N > 0 |
    pascal_data([data(N,Result)],1,[[1,0],[1,0]],1).

pascal_data([data(N,D)|S],N,Data,Max) :- true |
    D = Data,
    pascal_data(S,N,Data,Max).
pascal_data([data(N,D)|S],M,Data,Max) :- N =\= M,N =< Max |
    pascal_data(S,M,Data,Max).
pascal_data([data(N,D)|S],Max,Data,Max) :- Max < N,M1 := Max + 1 |
    new_pascal(M1,N,Data,D,S),
    pascal_data(S,Max,Data,N).
pascal_data([data(N,D)|S],M,Data,Max) :-  Max < N, M < Max |
    pascal_data(S,M,Data,N).
pascal_data([],_,_,_) :- true | true.

new_pascal(N,N,Data,D,Stream) :- true |
    make_pascal_data(Data,D),
    pascal_data(Stream,N,D,N).
new_pascal(N,M,Data,D,Stream) :- N < M, N1 := N+1 |
    make_pascal_data(Data,Data1),
    pascal_data(Stream,N,Data1,M),
    new_pascal(N1,M,Data1,D,Stream).

/* bignum version... */
make_pascal_data([F1,F2|Data],New) :- true |
    big_plus(Nf2,F1,F2),
    New = [F1,Nf2|New1],
    make_pascal_data([F2|Data],New1,[Nf2,F1]).

make_pascal_data([N],New,E) :- true | New = [N].
make_pascal_data([A,A|C],New,E) :- true |
    big_plus(B,A,A),
    New = [B|E].
otherwise.
make_pascal_data([A,B|C],New,E) :- true |
    big_grt(A,B,Status),
    make_pascal_data1(Status,A,B,C,New,E).
```

```
make_pascal_data1(yes,_,_,_,New,E) :- true | New = E.
make_pascal_data1( no,A,B,C,New,E) :- true |
    big_plus(D,A,B),
    New = [D|New1],
    make_pascal_data([B|C],New1,[D|E]).

/* normal version...
make_pascal_data([F1,F2|Data],New) :- Nf2 := F1+F2 |
    New = [F1,Nf2|New1],
    make_pascal_data([F2|Data],New1,[Nf2,F1]).
make_pascal_data([N],New,E) :- true | New = [N].
make_pascal_data([A,A|C],New,E) :- B := A+A | New = [B|E].
make_pascal_data([A,B|C],New,E) :- A > B | New = E.
make_pascal_data([A,B|C],New,E) :- A < B, D := A+B |
    New = [D|New1],
    make_pascal_data([B|C],New1,[D|E]).
*/

/*-------------------------------------------------------------------------
Program: Bignum for Pascal Benchmark
Author:  R. O'Keefe (translated to FGHC by E. Tick, revised by A. Okumura)
Date:    July 26 1988
*/

% this interface is meant to save storage...
big_plus(X,Y,Z) :- true |
    eval('+'(real(+,Y,[1]),real(+,Z,[1])),real(+,X,[1])).

big_grt(X,Y,Status) :- true |
    eval(real(+,X,[1]),A),
    eval(real(+,Y,[1]),B),
    comq(A, B, 100000, R),
    getstatus(R, '>', Status).

getstatus(X,X,Status) :- true | Status = yes.
otherwise.
getstatus(_,_,Status) :- true | Status = no.

eval('+'(X,Y), C) :- true |
    eval(X, A),
    eval(Y, B),
    addq(A, B, 100000, C).
otherwise.
eval(X,Y) :- true | X = Y.

%-------------------------------------------------------------------------

comq(A,A,_,S) :- true | S = '='.
otherwise.
comq(real('+',Na,Da), real('+',Nb,Db), R, S) :- true |
    muln(Na, Db, R, Xa),
    muln(Nb, Da, R, Xb),
    comn(Xa, Xb, '=', S).
comq(real('+',Na,Da), real('-',Nb,Db), R, S) :- true | S = '>'.
comq(real('-',Na,Da), real('+',Nb,Db), R, S) :- true | S = '<'.
comq(real('-',Na,Da), real('-',Nb,Db), R, S) :- true |
    muln(Na, Db, R, Xa),
    muln(Nb, Da, R, Xb),
    comn(Xb, Xa, '=', S).
comq(Na, real('+',Nb,Db), R, S) :- Na >= 0 |
    muln([Na], Db, R, Xa),
```

```
    comn(Xa, Nb, =, S).
comq(Na, real('-',Nb,Db), R, S) :- Na >= 0 | S = '>'.
comq(Na, real('+',Nb,Db), R, S) :- Na < 0 | S = '<'.
comq(Na, real('-',Nb,Db), R, S) :- Na < 0, Nz := (0-Na) |
    muln([Nz], Db, R, Xa),
    comn(Nb, Xa, '=', S).
comq(real('+',Na,Da), Nb, R, S) :- Nb >= 0 |
    muln([Nb], Da, R, Xb),
    comn(Na, Xb, '=', S).
comq(real('+',Na,Da), Nb, R, S) :- Nb < 0 | S = '>'.
comq(real('-',Na,Da), Nb, R, S) :- Nb >= 0 | S = '<'.
comq(real('-',Na,Da), Nb, R, S) :- Nb < 0, Nz := (0-Nb) |
    muln([Nz], Da, R, Xb),
    comn(Xb, Na, '=', S).
comq(Na, Nb, R, S) :- Na > Nb  | S = '>'.
comq(Na, Nb, R, S) :- Nb >= Na | S = '<'.

%-------------------------------------------------------------------------

addq(A, B, R, S) :- true |
    real(A, R, Sa, Na, Da),
    real(B, R, Sb, Nb, Db),
    muln(Na, Db, R, Xa),
    muln(Nb, Da, R, Xb),
    addz(Sa,Xa, Sb,Xb, R, Sc,Xc),
    gcdn(Xc, Da, R, _, Nx, Ya),
    gcdn(Nx, Db, R, _, Nc, Yb),
    muln(Ya, Yb, R, Dc),
    standardise(real(Sc, Nc, Dc),S).

%-------------------------------------------------------------------------

muln([], _, _, S) :- true | S = [].
muln(_, [], _, S) :- true | S = [].
otherwise.
muln(A, B, R, C) :- true | muln(A, B, [], R, C).

muln([], _, Ac, _, Out) :- true | Out = Ac.
muln([D1|T1], N2, Ac, R, Out) :- true |
    Out = [D3|Pr],
    mul1(N2, D1, R, P2),
    addn(Ac, P2, 0, R, Sm),
    conn1(D3, An, Sm),
    muln(T1, N2, An,R, Pr).

mul1(_, 0, _, Pr) :- true | Pr = [].
otherwise.
mul1(A, M, R, Pr) :- true |
    mul1(A, M, 0, R, Pr).

mul1([D1|T1], M, C, R, Out) :-
    D2 := (D1*M+C) mod R,
    Co := (D1*M+C) / R |
    Out = [D2|T2],
    mul1(T1, M, Co, R, T2).
mul1(     [], _, 0, _, Out) :- true | Out = [].
otherwise.
mul1(     [], _, C, _, Out) :- true | Out = [C].

%-------------------------------------------------------------------------

addz('+',A,'+',B,R,S,C) :- true | S = '+', addn(A, B, 0, R, C).
```

```
addz('+',A,'-',B,R,S,C) :- true |          subn(A, B, R, S, C).
addz('-',A,'+',B,R,S,C) :- true |          subn(B, A, R, S, C).
addz('-',A,'-',B,R,S,C) :- true | S = '-', addn(B, A, 0, R, C).

addn([D1|T1], [D2|T2], Cin, R, Out) :-
    Sum := D1+D2+Cin,
      X := Sum mod 262144 |
    add2(T1, T2, R, Out, Sum, X).
addn([], L, 0, _, Out) :- true | Out=L.
addn(L, [], 0, _, Out) :- true | Out=L.
addn([], L, 1, R, M)   :- true | add1(L, R, M).
addn(L, [], 1, R, M)   :- true | add1(L, R, M).

add1([M|T], R, Out) :- N := M+1 | add3(N,T,R,Out).
add1([],    _, Out) :- true | Out = [1].

add3(N,T,R,Out) :- N < R | Out = [N|T].
otherwise.
% Prolog is funny: it checks that R =:= N, but it should not fail!
add3(N,T,R,Out) :- true | add1(T,R,S), Out = [0|S].

add2(T1, T2, R, Out, Sum, X) :- X >= R, D3 := X-R |
    Out = [D3|T3],
    addn(T1, T2, 1, R, T3).
otherwise.
add2(T1, T2, R, Out, Sum, X) :- true |
    Out = [Sum|T3],
    addn(T1, T2, 0, R, T3).

%-----------------------------------------------------------------------------

gcdn([], [], R, O1, O2, O3) :- true | O1 = [], O2 = undefined, O3 = undefined.
gcdn([],  B, R, O1, O2, O3) :- B \= [] | O1=B, O2 = [], O3 = [1].
gcdn( A, [], R, O1, O2, O3) :- A \= [] | O1=A, O2 = [1], O3 = [].
otherwise.
gcdn([1], B, _, O1, O2, O3) :- true | O1 = [1], O2 = [1], O3 =   B.
gcdn(A, [1], _, O1, O2, O3) :- true | O1 = [1], O2 =   A, O3 = [1].
otherwise.
gcdn(A, B, R, D, M, N) :- true |      %  A, B > 1
    gcdn(A, B, R, D),
    divn(A, D, R, M, _),
    divn(B, D, R, N, _).

gcdn(A, B, R, D) :- true |      %  A, B >= 1
    comn(A, B, '=', S),
    gcdn(S, A, B, R, D).

gcdn('=', A, _, _, D) :- true | D = A.
gcdn('<',[], B, _, D) :- true | D = B.
gcdn('<', A, B, R, D) :- A \= [] |
    estg(B, A, R, E),
    muln(E, A, R, P),
    subn(B, P, R, _, M),
    gcdn(A, M, R, D).
gcdn('>', A,[], _, D) :- true | D = A.
gcdn('>', A, B, R, D) :- B \= [] |
    estg(A, B, R, E),
    muln(E, B, R, P),
    subn(A, P, R, _, M),
    gcdn(M, B, R, D).

estg(A, [B], R, E) :- true |
```

```
    div1(A, B, R, Q, X),
    estg1(X, B, Q, R, E).
otherwise.
estg([_|A], [_|B], R, E) :- true |
    estg(A, B, R, E).

estg1(X, B, Q, R, E) :- F := X*2,  F =< B | E = Q.
otherwise.
estg1(X, B, Q, R, E) :- true | add1(Q, R, E).

%-------------------------------------------------------------------------

% we know that this failure (division by zero) never occurs in benchmark.
% divn(A, [], R, _, _) :- !, fail. % division by 0 is undefined

divn(A,[1], R, A, X) :- true | X = [].   % a very common special case
divn(A,[B], R, Q, X) :- true |           % nearly as common a case
    div1(A, B, R, Q, Y),
    conn(Y, [], X).
otherwise.
divn(A,  B, R, Q, X) :-
    comn(A, B, '=', S),
    divn1(S, A, B, R, Q, X).

divn1('<', A, _, _, Q, X) :- true | Q = [], X = A.
divn1('=', _, _, _, Q, X) :- true | Q = [1], X = [].
divn1('>', A, B, R, Q, X) :- true | divm(A, B, R, Q, X).

% mode(+,+,-)
conn(0, [], Out) :- true | Out = [].
otherwise.
conn(D, T, Out) :- true | Out = [D|T].

% mode(-,-,+)
conn1(D, T,    []) :- true | D = 0, T = [].
conn1(D, T, [X|Y]) :- true | D = X, T = Y.

div1([], _, _, Q1, X1) :- true | Q1 = [], X1 = 0.
div1([D1|T1], B1, R, Q1, X1) :- true |
    div1(T1, B1, R, Q2, X2),
    div11(X2,R,D1,B1,Q2,Q1).

div11(X2,R,D1,B1,Q2,Q1) :-
    D2 := (X2*R+D1) / B1,
    X1 := (X2*R+D1) mod B1 |
    conn(D2, Q2, Q1).

% divm(A, B, R, Q, X) is called with A > B > R
divm([D1|T1], B, R, Q1, X1) :- true |
    divm(T1, B, R, Q2, X2),
    conn(D1, X2, T2),
    div2(T2, B, R, D2, X1),
    conn(D2, Q2, Q1).
divm([], B, R, O1, O2) :- true | O1 = [], O2 = [].

div2(A, B, R, Q, X) :- true |
    estd(A, B, R, E),
    chkd(A, B, R, E, 0, Q, P),
    subn(A, P, R, S, X).

estd([A0,A1,A2], [B0,B1], R, E) :-
    F := R/2, B1 >= F, G := (A2*R+A1)/B1 |
```

```
    E = G.
otherwise.
estd([A0,A1,A2], [B0,B1], R, E) :-
    L := (A2*R+A1)/(B1+1) |
    mul1([B0,B1],    L, R, P),
    subn([A0,A1,A2], P, R, S, N),
    estd(N, [B0,B1], R, M),
    estd1(L,M,E).
estd([A0,A1], [B0,B1], R, E) :-
    F := (A1*R+A0+1)/(B1*R+B0) |
    E = F.
estd([A0], _, _, E) :- true | E = 0.
otherwise.
estd([A0|Ar], [B0|Br], R, E) :- true |
    estd(Ar, Br, R, E).
estd([], _, _, E) :- true | E = 0.

estd1(L,M,E) :- F := L+M | E=F.

chkd(A, B, R, E, 3, _, _) :- true | true.
otherwise.
chkd(A, B, R, E, K, E, P) :- true |
    mul1(B, E, R, P),
    comn(P, A, '<', S),
    chkd1(S, A, B, R, E, K, Q, P).

chkd1('<', A, B, R, E, K, Q, P) :- true | true.
otherwise.
chkd1(_, A, B, R, E, K, Q, P) :-
    L := K+1, F := E-1 |
    chkd(A, B, R, F, L, Q, P).

%-----------------------------------------------------------------------------

subn(A, B, R, S, C) :- true |
    comn(A, B, '=', 0),
    subn(0, A, B, R, S, C).

subn('<', A, B, R, F, C) :- true |
    F = '-',
    subp(B, A, 0, R, D),
    prune(D, C).
subn('>', A, B, R, F, C) :- true |
    F = '+',
    subp(A, B, 0, R, D),
    prune(D, C).
subn('=', A, B, R, F, C) :- true |
    F = '+', C = [].

prune([], Out) :- true | Out = [].
prune([0|L], M ) :- true |
    prune(L, T),
    prune1(T, M).
otherwise.
prune([D|L], Out) :- true |
    Out = [D|M],
    prune(L, M).

prune1([], M) :- true | M = [].
otherwise.
prune1(T, M) :- true | M = [0|T].
```

```
subp([D1|T1], [D2|T2], Bin, R, M) :- S := D1-D2-Bin |
    subp1(S, T1, T2, R, M).
subp(L, [], 0, _, M) :- true | M = L.
subp(L, [], 1, R, M) :- true | sub1(L, R, M).

subp1(S, T1, T2, R, M) :- S >= 0 |
    M = [S|T3],
    subp(T1, T2, 0, R, T3).
otherwise.
subp1(S, T1, T2, R, M) :- D3 := S+R |
    M = [D3|T3],
    subp(T1, T2, 1, R, T3).

sub1([0|T], R, Out) :- K := R-1 | Out = [K|S], sub1(T, R, S).
otherwise.
sub1([N|T], _, Out) :- M := N-1 | Out = [M|T].

%-------------------------------------------------------------------------------
comn([D1|T1], [D2|T2], D, S) :- true |
    com1(D1, D2, D, N),
    comn(T1, T2, N, S).
comn([], [], D, S) :- true | S = D.
comn([],  L, D, S) :- L \= [] | S = '<'.
comn(L,  [], D, S) :- L \= [] | S = '>'.

com1(X, X, D, E) :-  true | E = D.
com1(X, Y, _, E) :- X < Y | E = '<'.
com1(X, Y, _, E) :- X > Y | E = '>'.

%-------------------------------------------------------------------------------

real(    undefined, R, O1, O2, O3) :- true |
    O1 = '+', O2 = [], O3 = [].
real(real(S, N, D), R, O1, O2, O3) :- true |
    O1 = S, O2 = N, O3 = D.
real(N, R, O1, L, O3) :- N >= 0 |
    O1 = '+', O3 = [1], binrad(N, R, L).
real(N, R, O1, L, O3) :-  N  < 0, M := (0-N) |
    O1 = '-', O3 = [1], binrad(M, R, L).

binrad(0, R, Out) :- true | Out = [].
otherwise.
binrad(N, R, Out) :- K := N/R, M := N mod R |
    Out = [M|T],
    binrad(K, R, T).

standardise(real('+',[N],[1]), Ans) :- true | Ans = N.
standardise(real('-',[N],[1]), Ans) :- F := (0-N) | Ans = F.
standardise(real(  S,  N, []), Ans) :- true | Ans = undefined.
standardise(real(  _, [],[1]), Ans) :- true | Ans = 0.
otherwise.
standardise(Number, Ans) :- true | Ans = Number.
```

B.4 Semigroup

```
/*-----------------------------------------------------------------------------
Program: Semigroup
Author:  N. Ichiyoshi
Date:    July 28 1988

Notes:
1. To run:
        ?- go(N).
the output N should be 309.
2. this version does NOT include generators in final count (c.f. Prolog)
-----------------------------------------------------------------------------*/
go(N) :- true |
    generators(Gens),
    go1(Gens, Out),
    count(Out, N).

go1(Gens, Out) :- true |
    gen_g(Gens, Gin, Fin, Gout, Fout),
    gen_gen(Gens, Gin, NGin),
    connect(Gout, Fin),
    ends(Fout, _, _, NGin, Gens, Out-[]).

count(L,N) :- true | count(L,0,N).
count([],M,N) :- true | M = N.
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% g(+Gin, +Fin, -Gout, -Fout, +E)
%
%       Gin         : input generator stream
%       Fin         : input filter stream
%       Gout         : output generator stream
%       Fout         : output filter stream
%       E        : element (self)
%
g([gen(X,P,P0)|Gin1], Fin, Gout, Fout, E) :- true |
    mult(E, X, EX),
    P0 = [EX|P1],
    Gout = [gen(X,P,P1)|Gout1],
    g(Gin1, Fin, Gout1, Fout, E).
g([begin|Gin1], Fin, Gout, Fout, E) :- true |
    Gout = [begin|Gout1],
    g(Gin1, Fin, Gout1, Fout, E).
g([end|Gin1], Fin, Gout, Fout, E) :- true |
    Gout = [end|Gin1],
    f(Fin, Fout, E).

% f(+Fin,-Fout, +E)
%
%       filters out E from stream Fin to give Fout as result
%
%       Fin        : input stream of elements to be filtered
%       Fout        : output stream of elements filtered
%       E        : element (self)
%
f([X|Fin1], Fout, E) :- X = E | f(Fin1,Fout, E).
f([], Fout, E) :-       true  |        Fout = [].
otherwise.
f([X|Fin1], Fout, E) :- true  | Fout = [X|Fout1],    f(Fin1, Fout1, E).

% gen_g(+Xs, -G0, -F0, +G, +F)
```

```
%
%       creates NS1.
%
%       Xs        : list of elements
%       G0        : input generator stream
%       F0        : input filter stream
%       G        : output generator stream
%       F        : output filter stream
%
gen_g([X|Xs], G0,F0,G,F) :- true |
    g(G0,F0,G1,F1, X),
    gen_g(Xs, G1,F1,G,F).
gen_g([], G0,F0,G,F) :- true |
    G0 = G,
    F0 = F.

% ends(+Fout,+Gout,-Gin,-OGin, +Gen, -Out)
%
%       Fout        : output filter stream                  ... (NSn*G)\Sn
%       Gout        : output generator stream          ... NSn*G
%       Gin        : input generator stream
%       OGin        : old input generator stream
%       Gen        : list of original generators         ... G
%      Out     : output D-list stream
%
ends([begin,end|_], _, Gin, OGin, _, O1-O2) :- true | Gin=[], OGin=[], O1=O2.
ends([begin,X|Fout2], _, _, OGin, Gens, Out) :- X \= end |
    gen_gen(Gens,NGin0,NGin),
    ends([X|Fout2],NGin0,NGin,OGin, Gens, Out).
ends([end|Fout1], Gout, Gin, OGin, Gens, Out) :- true |
    connect(Gout, OGin),
    ends(Fout1, _, _, Gin, Gens, Out).
ends([X|Fout1], Gout, Gin, OGin, Gens, O1-O3) :- X \= begin, X \= end |
    O1 = [X|O2],    %  ET 3-8-88: output...
    g(Gout, Fout1, NewGout, NewFout, X),
    ends(NewFout, NewGout, Gin, OGin, Gens, O2-O3).

% gen_gen(+Gens, +G0,?G)
%
%       creates d-list representation of {}*G (see Notes on Representation).
%
%       Gens        : set of generators g1, ..., gK.
%       G0-G        : d-list to represent the list
%                       [gen(g1,G10,G1), ..., gen(gK,GK0,GK), end]
%
gen_gen(Gens, G0,G) :- true |
    G0 = [begin|G1],
    gen_gen1(Gens, G1,G).

gen_gen1([X|Xs], G0,G) :- true |
    G0 = [gen(X,P,P)|G1],
    gen_gen1(Xs, G1,G).
gen_gen1([], G0,G) :- true |
    G0 = [end|G].

% connect(+G, -F)        : connects gen-stream to filter stream
%
%       (see Notes on Representation (3) )
%
%       G        : stream of d-lists representing generated elements
%       F        : stream of generated elements to be filtered
%
```

```
connect([gen(_,P0,P)|G1], F) :- true |
    F = P0,
    connect(G1, P).
connect([begin|G1], F) :- true |
    F = [begin|F1],
    connect(G1, F1).
connect([end|G1], F) :- true |
    F = [end|G1].

% mult(+X, +Y, -Z)
%       X       : X = [X1, X2, ...]
%       Y       : Y = [Y1, Y2, ...]
%       Z       : Z = X*Y = [X1*Y1, X2*Y2, ...]
%
mult([X|Xs], [Y|Ys], Out) :- true |
    Out = [Z|Zs],
    m(X, Y, Z),
    mult(Xs, Ys, Zs).
mult([], [], Z) :- true | Z=[].

%-----------------------------------------------------------------------
%       Multiplication Rule for the Direct Product of
%       the 5-element Bradt Semigroup B2
%-----------------------------------------------------------------------

%       Bradt semigroup B2 = { 0, e, f, a, b }.
%
%       Multiplication table for B2 :
%
%              | 0   e   f   a   b
%            ---+-----------------
%             0 | 0   0   0   0   0
%             e | 0   e   0   a   0
%             f | 0   0   f   0   b
%             a | 0   0   a   0   e
%             b | 0   b   0   f   0

m(0,_,Z):- true | Z=0.
m(e,0,Z):- true | Z=0.
m(f,0,Z):- true | Z=0.
m(a,0,Z):- true | Z=0.
m(b,0,Z):- true | Z=0.
m(e,e,Z):- true | Z=e.
m(f,e,Z):- true | Z=0.
m(a,e,Z):- true | Z=0.
m(b,e,Z):- true | Z=b.
m(e,f,Z):- true | Z=0.
m(f,f,Z):- true | Z=f.
m(a,f,Z):- true | Z=a.
m(b,f,Z):- true | Z=0.
m(e,a,Z):- true | Z=a.
m(f,a,Z):- true | Z=0.
m(a,a,Z):- true | Z=0.
m(b,a,Z):- true | Z=f.
m(e,b,Z):- true | Z=0.
m(f,b,Z):- true | Z=b.
m(a,b,Z):- true | Z=e.
m(b,b,Z):- true | Z=0.

% 309+4 solutions...
generators(IL) :- true |
    IL = [
```

```
  [0,0,0,0,0, e,e,e,e,e, f,f,f,f,f, a,a,a,a,a, b,b,b,b,b, f,f,f,f,f,
   b,b,b,b,b, a,a,a,a,a],
  [0,e,f,a,b, 0,e,f,a,b, 0,e,f,a,b, 0,e,f,a,b, 0,e,f,a,b, 0,e,f,a,b,
   0,f,e,a,b, 0,e,f,a,b],
  [0,0,0,0,0, e,e,e,e,e, f,f,f,f,f, b,b,b,b,b, a,a,a,a,a, e,e,e,e,e,
   a,a,a,a,a, f,f,f,f,f],
  [0,e,f,b,a, 0,e,f,b,a, 0,e,f,b,a, 0,e,f,b,a, 0,e,f,b,a, 0,e,f,a,b,
   0,e,f,b,a, 0,e,f,b,a]
].
```

B.5 Queens

B.5.1 AOqueen

```
/*-------------------------------------------------------------------------
Program: N-Queens using layered-streams
Author:  A. Okumura
Date:    June 17 1988

1. To run: ?- go(M,N).
for example, when input M=8, output N=92 (number of solutions).
2. This version uses [X|Y] representation of layered-stream.
-------------------------------------------------------------------------*/
:- module queen.
:- public go/2.

go(M,N) :- true | queen(1,M,begin,A), count(A,N).

count(L,N) :- true | count(L,0,N).
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).
count([],M,N) :- true | M = N.

queen(I,N,In,Out) :- I < N,
    I1 := I+1 |
    q(1,N,In,Mid),
    queen(I1,N,Mid,Out).
queen(N,N,In,Out) :- true | lastQ(1,N,In,Out).

q(I,N,In,Out) :- I =< N,
    I1 := I+1 |
    filter(In,I,1,Out1),
    q(I1,N,In,Outs),
    Out=[[I|Out1]|Outs].
q(I,N,_,Out) :- I > N | Out = [].

lastQ(I,N,In,Out) :- I =< N,
    I1 := I+1 |
    lastFilter([I],In,I,1,Out,Out1),
    lastQ(I1,N,In,Out1).
lastQ(I,N,_,Out) :- I > N | Out = [].

filter(begin,_,_,Out) :- true | Out = begin.
filter([],_,_,Out) :- true    | Out = [].
filter([[I|_]|Ins],I,D,Out) :- true      | filter(Ins,I,D,Out).
filter([[J|_]|Ins],I,D,Out) :- D =:= I-J | filter(Ins,I,D,Out).
filter([[J|_]|Ins],I,D,Out) :- D =:= J-I | filter(Ins,I,D,Out).
otherwise.
filter([[J|In1]|Ins],I,D,Out) :- D1 := D+1 |
    filter(In1,I,D1,Out1),
    filter(Ins,I,D,Outs),
    Out = [[J|Out1]|Outs].

lastFilter(Stack,begin,_,_,S,T) :- true | S = [Stack|T].
lastFilter(Stack,[],_,_,S,T) :- true | S = T.
lastFilter(Stack,[[I|_]|Ins],I,D,S,T) :- true | lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[[J|_]|Ins],I,D,S,T) :- D =:= I-J |
    lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[[J|_]|Ins],I,D,S,T) :- D =:= J-I |
    lastFilter(Stack,Ins,I,D,S,T).
otherwise.
lastFilter(Stack,[[J|In]|Ins],I,D,S,T) :- D1 := D+1 |
    lastFilter([J|Stack],In,I,Di,S,SS),
    lastFilter(Stack,Ins,I,D,SS,T).
```

B.5.2 KKqueen

```
/*-------------------------------------------------------------------------
Program: N-Queens (all-solutions AND-parallel)
Author:  K. Kumon
Date:    May 18 1988

Notes:
1. To run:
    ?- go(N,M).
for example, when input N=9, output M=352 (number of solutions).
-------------------------------------------------------------------------*/
go(N,M) :- true |
    gen(N,L),
    queen(L,[],[],X,[]),
    count(X, M).

queen([P|U], C, L, I0, I2):- true |
    append(U, C, N),
    check(L, P, 1, N, L, I0, I1),
    queen(U, [P|C], L, I1, I2).
queen([], [], L, I, 0):- true | I=[L|0].
queen([], [_|_], _, I, 0):- true | I=0.

check([],    T, D, N, B, I, 0):- true |
    queen(N, [], [T|B], I, 0).
check([P|_], T, D, N, B, I, 0):- T=:=P+D | I=0.
check([P|_], T, D, N, B, I, 0):- T=:=P-D | I=0.
otherwise.
check([P|R], T, D, N, B, I, 0):- D1:=D+1 |
    check(R, T, D1, N, B, I, 0).

gen(N, X) :- N>0, M := N-1 | X = [N|Xs], gen(M,Xs).
gen(N, X) :- N=:=0 | X = [].

append([A|X],Y,Z):- true | Z=[A|Z1], append(X,Y,Z1).
append([],   Y,Z):- true | Z=Y.

count(L,N) :- true | count(L,0,N).
count([],M,N) :- true | M = N.
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).
```

B.5.3 KUqueen

```
/*-------------------------------------------------------------------------
Program: N-Queens (translated from Prolog MBqueen)
Author:  K. Ueda
Date:    May 18 1988

Notes:
1. To run:
    ?- go(N,M).
for example, when input N=8, output is M=92 (number of solutions).
-------------------------------------------------------------------------*/
go(N,M) :- true |
    gen(N,L),
    queen(L,[],'L1',X,[]),
    count(X, M).

gen(N, X) :- N>0, M := N-1 | X = [N|Xs], gen(M,Xs).
gen(N, X) :- N=:=0 | X = [].

count(L,N) :- true | count(L,0,N).
count([],M,N) :- true | M = N.
count([X|Xs],M,N) :- M1 := M+1 | count(Xs,M1,N).

queen([H|T],R,Cont,Rs0,Rs1) :- true |
    select([H|T],'L2'(Cont,R),'L2',Rs0,Rs1).
queen([],R,Cont,Rs0,Rs1) :- true | Rs0 = [R|Rs1].

select(HT,Cont,Conts,Rs0,Rs2) :- true |
    d1(HT,Cont,Conts,Rs0,Rs1),
    d2(HT,Cont,Conts,Rs1,Rs2).

d1([A|L],'L2'(Cont,R),Conts,Rs0,Rs1) :- true |
    check(R,A,1,'L2b'(Cont,R,A,L,Conts),Rs0,Rs1).
d1([],   Cont,Conts,Rs0,Rs1) :- true | Rs0=Rs1.

d2([H|T],Cont,Conts,Rs0,Rs1) :- true |
    select(T,Cont,'L5'(Conts,H),Rs0,Rs1).
d2([],   Cont,Conts,Rs0,Rs1) :- true | Rs0=Rs1.

check([],   U,N,'L2b'(Cont,R,A,L,Conts),Rs0,Rs1) :- true |
     b(Conts,'L3'(Cont,R,A),L,Rs0,Rs1).
check([H|T],U,N,Cont,Rs0,Rs1) :- H =:= U+N | Rs0=Rs1.
check([H|T],U,N,Cont,Rs0,Rs1) :- H =:= U-N | Rs0=Rs1.
otherwise.
check([H|T],U,N,Cont,Rs0,Rs1) :- N1:=N+1 |
    check(T,U,N1,Cont,Rs0,Rs1).

b('L5'(Conts,A),Cont,T,Rs0,Rs1) :- true |
    b(Conts,Cont,[A|T],Rs0,Rs1).
b('L2','L3'(Cont,R,A),L,Rs0,Rs1) :- true |
    queen(L,[A|R],Cont,Rs0,Rs1).
```

C Appendix: Sample Cache Simulator Output

In this section listings are given of the cache simulator output for the five major benchmarks. See Section 5.3.3 for an explanation of how to interpret this raw data. A single I+D, 1K word cache (256 columns) simulation is shown for each program. The parameters of all simulations are identical and shown only for the first benchmark. Note that the KL1 system does not count `META` and `ETC` (miscellaneous) references. If there are no bus collisions, the table is not included.

Triangle - Aurora - 8 PEs
Cache params: c256,s4,w4,l1

GVNPTCL 00000001, GVNSMOD 00000001, BYTXMD 00000002, AMBORAM 00000001
GVNPE 00000008, GVNSET 00000004, GVNCOL 00000100, GVNBLK 00000004
GVNSECT 00000001, BUSWIDT 00000001, MACCTIM 00000008, CYCXTIM 00000001
INVTIME 00000002, GOALRPS 000000c8, GOALCYC 00000032

TABLE GIVEN CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
R	4578021	5637437	225489	5726550	41878	41016	1186713	17437104
W	16658	0	17326	761227	38133	1138895	1182075	3154314
DW	334092	0	0	0	0	0	0	334092
LR	0	0	0	1950	0	0	0	1950
UW	0	0	0	0	0	0	0	0
U	0	0	0	1950	0	0	0	1950
TOTAL	4928771	5637437	242815	6491677	80011	1179911	2368788	20929410

TABLE ISSUED BUS COMMAND(AREA)

BUSCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
F	394	4806	295	2832	33	224	412	8996
FI	10	0	17	1557	16	74	164	1798
IV	10	0	4	737	0	0	31	782
TOTAL	414	4806	316	5126	49	258	607	11576

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	60	293	18	10	3	37	9	430
13:FROM-GM-ONLY	182	1582	245	427	46	221	249	2952
10:MCTOC-SOUT	10	0	3	25	0	0	4	42
07:MCTOC-ONLY	121	0	38	1670	0	0	72	1901
10:CCTOC-SOUT	9	125	0	20	0	0	22	176
07:CCTOC-ONLY	22	2806	8	2237	0	0	220	5293
05:SOUT-ONLY	243	0	0	0	0	0	0	243
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	10	0	4	737	0	0	31	782
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	657	4806	316	5126	49	258	607	11819

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	780	3809	234	130	39	481	117	5590
13:FROM-GM-ONLY	2366	20566	3185	5551	598	2873	3237	38376
10:MCTOC-SOUT	100	0	30	250	0	0	40	420
07:MCTOC-ONLY	847	0	266	11690	0	0	504	13307
10:CCTOC-SOUT	90	1250	0	200	0	0	220	1760
07:CCTOC-ONLY	154	19642	56	15659	0	0	1540	37051
05:SOUT-ONLY	1215	0	0	0	0	0	0	1215
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	20	0	8	1474	0	0	62	1564
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	5572	45267	3779	34954	637	3354	5720	99283

TABLE PREVIOUS STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	87927	11659341	5671320	9520	17428108	8996	17437104
W	327	3403454	194	117	3404092	610	3404702
LW	5	82403	0	16	82425	1263	83688
LR	0	291	365	106	762	1188	1950
UW	0	0	0	0	0	0	0
U	0	1240	0	710	1950	0	1950
TOTAL	88260	15146729	5671879	10469	20917337	12057	20929394

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	1943	4276	6219	2777	8996
W	0	5	5	605	610
DW	0	0	0	1263	1263
LR	0	1188	1188	0	1188
UW	0	0	0	0	0
U	0	0	0	0	0
TOTAL	1943	5469	7412	4645	12057

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	334092
ISSUED	83688
WITHOUT-SWAP-OUT	1020
WITH-SWAP-OUT	243

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

1020	DW-WITHOUT-SOUT
20918357	T-HIT + DW-WITHOUT-SOUT
11037	T-MISS - DW-WITHOUT-SOUT
99.95 [%]	HIT-RATIO
0.05 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
916	1202	3002	112	0	135	2825	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	INVALID	TOTAL
956	3094	165	521	41	96	359	2960	8192

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	20929410	99283	0.005
1	7	20929410	95901	0.005
1	6	20929410	92519	0.004
1	5	20929410	89137	0.004
1	0	20929410	74377	0.004
2	8	20929410	76773	0.004
2	7	20929410	73391	0.004
2	6	20929410	70009	0.003
2	5	20929410	66627	0.003
2	0	20929410	51007	0.002

Semigroup - Aurora - 8 PEs
Cache parms: c256,s4,w4,t1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
R	1763728	3645525	1064500	3407667	69170	330496	232525	10513611
W	207236	0	107570	297037	27269	27720	55130	721962
DW	341808	0	0	0	0	0	0	341808
LR	0	0	0	44331	0	0	0	44331
UW	0	0	0	0	0	0	0	0
U	0	0	0	44331	0	0	0	44331
TOTAL	2312772	3645525	1172070	3793366	96439	358216	287655	11666043

TABLE ISSUED BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
F	63646	35611	6891	425160	1219	14384	11280	558191
FI	88	0	27	34549	739	846	1404	37653
IV	138	0	391	24698	0	0	2086	27313
TOTAL	63872	35611	7309	484407	1958	15230	14770	623157

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	809	682	158	485	250	1575	191	4151
13:FROM-GM-ONLY	10414	7510	2655	21427	1708	13654	1485	58853
10:MCTOC-SOUT	1343	0	154	1863	0	0	273	3633
07:MCTOC-ONLY	8025	0	1068	55296	0	0	4456	68835
10:CCTOC-SOUT	5091	2913	325	14427	0	0	592	23348
07:CCTOC-ONLY	38052	24506	2558	366221	0	0	5687	437024
05:SOUT-ONLY	1096	0	0	0	0	0	0	1096
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	138	0	391	24698	0	0	2086	27313
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	64968	35611	7309	484407	1958	15230	14770	624253

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	10517	8866	2054	6305	3250	20488	2483	53963
13:FROM-GM-ONLY	135382	97630	34515	278551	22204	177502	19305	765089
10:MCTOC-SOUT	13430	0	1540	18630	0	0	2730	36330
07:MCTOC-ONLY	56175	0	7476	387082	0	0	31192	481845
10:CCTOC-SOUT	50910	29130	3250	144270	0	0	5920	233480
07:CCTOC-ONLY	266364	171542	17906	2563547	0	0	39809	3059168
05:SOUT-ONLY	5480	0	0	0	0	0	0	5480
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	276	0	782	49396	0	0	4172	54626
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	518534	307168	67523	3447701	25454	197990	105611	4689981

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	612902	2400158	6668621	273739	9955420	558191	10513611
W	5304	941922	6737	10966	967929	10165	978094
DW	3	68049	0	0	68052	17624	85676
LR	8	7225	7567	2043	16843	27488	44331
UW	0	0	0	0	0	0	0
U	203	30253	63	13271	43792	539	44331
TOTAL	618420	3450607	6682988	300021	11052036	614007	11666043

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	72468	431933	504401	53790	558191
W	0	1281	1281	8884	10165
DW	0	0	0	17624	17624
LR	0	27158	27158	330	27488
UW	0	0	0	0	0
U	0	0	0	539	539
TOTAL	72468	460372	532840	81167	614007

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	341808
ISSUED	85676
WITHOUT-SWAP-OUT	16528
WITH-SWAP-OUT	1096

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

16528	DW-WITHOUT-SOUT
11068564	T-HIT + DW-WITHOUT-SOUT
597479	T-MISS - DW-WITHOUT-SOUT
94.88 [%]	HIT-RATIO
5.12 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
764	788	6156	318	0	166	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	INVALID	TOTAL
2907	732	92	3365	47	544	339	166	8192

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	11666043	4689981	0.402
1	7	11666043	4626977	0.397
1	6	11666043	4563973	0.391
1	5	11666043	4500969	0.386
1	0	11666043	4206704	0.361
2	8	11666043	3442139	0.295
2	7	11666043	3379135	0.290
2	6	11666043	3316131	0.284
2	5	11666043	3253127	0.279
2	0	11666043	2950560	0.253

Puzzle Answer 8 PEs
Cache param: c256,s4,w4,l1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
R	1908750	2184800	163566	1440931	33099	601789	1136437	7478372
W	13020	0	12755	907931	30742	586403	1128007	2678858
DW	153204	0	0	0	0	0	0	153204
LR	0	0	0	2557	0	0	0	2557
UW	0	0	0	0	0	0	0	0
U	0	0	0	2557	0	0	0	2557
TOTAL	2074974	2184800	176321	2362976	63841	1188192	2264444	10315548

TABLE ISSUED-BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
F	1687	4145	291	3932	30	506	1042	11633
FI	19	0	26	2073	14	155	371	2658
IV	27	0	8	1119	0	0	123	1277
TOTAL	1733	4145	325	7124	44	661	1536	15568

TABLE BUS-USE-TYPE(OPERATION)

CYCLE-PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	6	209	13	14	6	90	85	423
13:FROM-GM-ONLY	77	1531	225	594	38	571	720	3756
10:MCTOC-SOUT	320	0	6	50	0	0	23	399
07:MCTOC-ONLY	1344	0	48	2526	0	0	215	3953
10:CCTOC-SOUT	38	427	8	61	0	0	44	580
07:CCTOC-ONLY	121	1978	17	2758	0	0	306	5180
05:SOUT-ONLY	241	0	0	0	0	0	0	241
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	27	0	8	1119	0	0	123	1277
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	1974	4145	325	7124	44	661	1536	15809

TABLE BUS-USE-TYPE(CYCLE)

CYCLE-PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	78	2717	169	192	78	1170	1105	5499
13:FROM-GM-ONLY	1001	19903	2925	7722	494	7423	9360	48828
10:MCTOC-SOUT	3200	0	60	500	0	0	230	3990
07:MCTOC-ONLY	8008	0	336	17682	0	0	1645	27671
10:CCTOC-SOUT	380	1270	80	630	0	0	440	5800
07:CCTOC-ONLY	847	13846	119	19306	0	0	2142	36260
05:SOUT-ONLY	1205	0	0	0	0	0	0	1205
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	54	0	16	2238	0	0	246	2554
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	14773	40736	3705	48260	572	8593	15168	131807

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	66715	3665908	3088925	645191	7466739	11633	7478372
W	797	2791110	311	373	2792591	1141	2793732
DW	1	77156	1	13	37171	1146	38317
LR	2	445	466	127	1040	1517	2557
UW	0	0	0	0	0	0	0
U	10	1718	0	825	2553	4	2557
TOTAL	67525	6496337	3089703	646529	10300094	15441	10315535

TABLE MISS ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	4352	4227	8579	3054	11633
W	0	24	24	1117	1141
DW	0	0	0	1146	1146
LR	0	1509	1509	8	1517
UW	0	0	0	0	0
U	0	0	0	4	4
TOTAL	4352	5760	10112	5329	15441

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	153204
ISSUED	38318
WITHOUT-SWAP-OUT	905
WITH-SWAP-OUT	241

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

905	DW-WITHOUT-SOUT
10300999	T-HIT + DW-WITHOUT-SOUT
14535	T-MISS - DW-WITHOUT-SOUT
99.86 [%]	HIT-RATIO
0.14 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
994	1247	2493	247	0	199	3012	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	INVALID	TOTAL
1195	2077	130	721	37	282	539	3211	8192

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	10315548	131807	0.013
1	7	10315548	127628	0.012
1	6	10315548	123449	0.012
1	5	10315548	119270	0.012
1	0	10315548	100490	0.010
2	8	10315548	100785	0.010
2	7	10315548	96606	0.009
2	6	10315548	92427	0.009
2	5	10315548	88248	0.009
2	0	10315548	68622	0.007

Pascal - Aurora - 8 PEs
Cache parms: c256,s4,w4,t1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
R	2519958	3493920	1455308	28924213	218046	248304	1504766	38364515
W	112142	0	173893	1926302	100667	135857	1182513	3631374
DW	716908	0	0	0	0	0	0	716908
LR	0	0	0	190819	0	0	0	190819
UW	0	0	0	0	0	0	0	0
U	0	0	0	190819	0	0	0	190819
TOTAL	3349008	3493920	1629201	31232153	318713	384161	2687279	43094435

TABLE ISSUED-BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
F	65902	144798	9026	759476	2413	18714	7238	1007567
FI	240	0	903	120324	83	1015	7779	130344
IV	132	0	523	78776	0	0	1144	80575
TOTAL	66274	144798	10452	958576	2496	19729	16161	1218486

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	3030	24701	941	998	731	4298	2179	36878
13:FROM-GM-ONLY	13006	75531	5208	16286	1765	15431	6714	133941
10:MCTOC-SOUT	4658	0	129	2456	0	0	311	7554
07:MCTOC-ONLY	15256	0	665	242536	0	0	2088	260545
10:CCTOC-SOUT	8136	9070	625	17328	0	0	834	35993
07:CCTOC-ONLY	22056	35496	2361	600196	0	0	2891	663000
05:SOUT-ONLY	11294	0	0	0	0	0	0	11294
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	132	0	523	78776	0	0	1144	80575
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	77568	144798	10452	958576	2496	19729	16161	1229780

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
13:FROM-GM-SOUT	39390	321113	12233	12974	9503	55874	28327	479414
13:FROM-GM-ONLY	169078	981903	67704	211718	22945	200603	87282	1741233
10:MCTOC-SOUT	46580	0	1290	24560	0	0	3110	75540
07:MCTOC-ONLY	106792	0	4655	1697752	0	0	14616	1823815
10:CCTOC-SOUT	81360	90700	6250	173280	0	0	8340	359930
07:CCTOC-ONLY	154392	248472	16527	4201372	0	0	20237	4641000
05:SOUT-ONLY	56470	0	0	0	0	0	0	56470
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	264	0	1046	157552	0	0	2288	161150
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	654326	1642188	109705	6479208	32448	256477	164200	9338552

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	1041584	5501954	29732466	1079467	37355471	1007567	38363038
W	22272	4102755	4510	14599	4144136	15520	4159656
DW	4308	113469	3342	7442	128561	50759	179320
LR	27	14502	45092	16374	75995	114824	190819
UW	0	0	0	0	0	0	0
U	372	82012	255	106189	188828	1991	190819
TOTAL	1068563	9814692	29785665	1224071	41892991	1190661	43083652

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	268099	584038	852137	155430	1007567
W	0	878	878	14642	15520
DW	0	0	0	50759	50759
LR	0	114077	114077	747	114824
UW	0	0	0	0	0
U	0	0	0	1991	1991
TOTAL	268099	698993	967092	223569	1190661

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	716908
ISSUED	179320
WITHOUT-SWAP-OUT	39465
WITH-SWAP-OUT	11294

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

39465	DW-WITHOUT-SOUT
41932456	T-HIT + DW-WITHOUT-SOUT
1151196	T-MISS - DW-WITHOUT-SOUT
97.33 [%]	HIT-RATIO
2.67 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
1217	2662	3864	90	0	359	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	INVALID	TOTAL
4062	1102	104	1949	28	419	169	359	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	ENV	NODE	LBA	GBA	TRAIL	TOTAL
0	0	0	1	0	0	0	1

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REP	BUS-CYCLE	TRAFFIC-RATIO
1	8	43094435	9338552	0.217
1	7	43094435	9167733	0.213
1	6	43094435	8996914	0.209
1	5	43094435	8826095	0.205
1	0	43094435	8156390	0.189
2	8	43094435	6953048	0.161
2	7	43094435	6782229	0.157
2	6	43094435	6611410	0.153
2	5	43094435	6440591	0.149
2	0	43094435	5697130	0.132

```
HKeeper    Aurora - 8 PEs
Cache param: c2w,c4,w4,t1

TABLE GIVEN CPU COMMAND(AREA)
GIVNCMD      HEAP      INST       ENV      NODE       LBA       GBA     TRAIL     TOTAL
R         1922845   3029128   1321129   2390809    534174    593317    707955  10498457
W          221297         0    108178   1924652    299559    356326    678400   3588403
DW         756659         0         0         0         0         0         0    756659
LR              0         0         0     10626         0         0         0     10626
UW              0         0         0         0         0         0         0         0
U               0         0         0     10626         0         0         0     10626
TOTAL     2900801   3029128   1429307   4336713    833724    949643   1385455  14864771

TABLE ISSUED BUS COMMAND(AREA)
BUSCMD       HEAP      INST       ENV      NODE       LBA       GBA     TRAIL     TOTAL
F            2551      5945       355     18172        62       591       539     28215
FI             13         0        16      7105        46        51       297      7528
IV             11         0         2      4143         0         0        74      4230
TOTAL        2575      5945       373     29420       108       642       910     39973

TABLE BUS USE TYPE(OPERATION)
CYCLE:PATTERN          HEAP      INST       ENV      NODE       LBA       GBA     TRAIL     TOTAL
13:FROM-GM-SOUT         456      2069        12        11        10       119        80      2757
13:FROM-GM-ONLY        1060      3719       281       712        98       523       231      6624
10:MCTOC-SOUT           160         0         4        90         0         0        20       274
07:MCTOC-ONLY           811         0        47     12488         0         0       231     13577
10:CCTOC-SOUT            21        12         8       166         0         0        53       260
07:CCTOC-ONLY            56       145        19     11910         0         0       221     12251
05:SOUT-ONLY           2164         0         0         0         0         0         0      2164
05:SOUT-EXTRA             0         0         0         0         0         0         0         0
02:INV-ONLY              11         0         2      4143         0         0        74      4230
05:FLUSH-BACK             0         0         0         0         0         0         0         0
05:FLUSH-EXTRA            0         0         0         0         0         0         0         0
   TOTAL               4739      5945       373     29420       108       642       910     42137

TABLE BUS USE TYPE(CYCLE)
CYCLE:PATTERN          HEAP      INST       ENV      NODE       LBA       GBA     TRAIL     TOTAL
13:FROM-GM-SOUT        5928     26897       156       143       130      1547      1040     35841
13:FROM-GM-ONLY       13780     48347      3653      9256      1274      6799      3003     86112
10:MCTOC-SOUT          1600         0        40       900         0         0       200      2740
07:MCTOC-ONLY          5677         0       329     87416         0         0      1617     95039
10:CCTOC-SOUT           210       120        80      1660         0         0       530      2600
07:CCTOC-ONLY           392      1015       133     82670         0         0      1547     85757
05:SOUT-ONLY          10820         0         0         0         0         0         0     10820
05:SOUT-EXTRA             0         0         0         0         0         0         0         0
02:INV-ONLY              22         0         4      8286         0         0       148      8460
05:FLUSH-BACK             0         0         0         0         0         0         0         0
05:FLUSH-EXTRA            0         0         0         0         0         0         0         0
   TOTAL              38429     76379      4395    190331      1404      8346      8085    127369

TABLE PREVIOUS-STATE(AREA) ALL. : ALL-AREA
CPUCMD         EC         EM         SC         SM      T-HIT     T-MISS      TOTAL
R          347958    6211322    3875058      35904  10470242      28215   10498457
W             609    4152412        545       1092   4154658       1119    4155777
IW              0     184129          0          5    184134       5146     189280
LR              0       1624       2276        317      4217       6409      10626
UW              0          0          0          0         0          0          0
U               1       6722          0       3894     10617          9      10626
TOTAL      348568   10556209    3877879      41212  14823868      40898   14864766

TABLE MISS-ANALYSIS(AREA) ALL. : ALL-AREA
CPUCMD       FRMC       FRCC    FRCACHE    FROM-GM     T-MISS
R           13851       6073      19924       8291      28215
W               0         30         30       1089       1119
DW              0          0          0       5146       5146
LR              0       6408       6408          1       6409
UW              0          0          0          0          0
U               0          0          0          9          9
TOTAL       13851      12511      26362      14536      40898

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL. : ALL-AREA
GIVEN                  756659
ISSUED                 189280
WITHOUT-SWAP-OUT         2982
WITH SWAP-OUT            2164

TABLE CACHE-HIT-RATIO(AREA) ALL. : ALL-AREA
            2982          DW-WITHOUT-SOUT
        14826850          T-HIT + DW-WITHOUT-SOUT
           37916          T-MISS - DW-WITHOUT-SOUT
           99.74 [%] HIT-RATIO
            0.26 [%] MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot after execution
      EC         EM         SC         SM          C          I     UNUSED      TOTAL
    1282       1070       1578        333          0        291       3638       8192

TABLE CACHE-DIRECTORY-AREA Snapshot after execution
    HEAP       INST        ENV       NODE        LBA        GBA      TRAIL    INVALID      TOTAL
    1646        553        172       1116         54        304        418       3929       8192

TABLE BUS-TRAFFIC-RATIO
BUS-WIDTH[W] MEM-ACC-TIME    MEM-REF BUS-CYCLE TRAFFIC-RATIO
           1            8   14864771    327369         0.022
           1            7   14864771    317988         0.021
           1            6   14864771    308607         0.021
           1            5   14864771    299226         0.020
           1            0   14864771    266106         0.018
           2            8   14864771    250487         0.017
           2            7   14864771    241106         0.016
           2            6   14864771    231725         0.016
           2            5   14864771    222344         0.015
           2            0   14864771    189710         0.012
```

Triangle ALL 8 PEs
Cache parms: c256,s4,w4,l1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
R	926172	11371781	4057090	60	0	418975	0	18774078
W	59110	0	4057025	59	0	418985	0	4535179
DW	970384	0	0	0	0	0	0	970384
LR	1687602	0	0	0	0	198561	0	1886163
UW	575847	0	0	0	0	198561	0	774408
U	1146123	0	0	0	0	0	0	1146123
TOTAL	5365238	11371781	8114115	119	0	1234082	0	28086335

TABLE ISSUED-BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
F	81320	127934	231811	21	0	248486	0	691572
FI	179563	0	141165	5	0	227430	0	548163
IV	55300	0	22752	13	0	240905	0	318970
TOTAL	316183	127934	397728	39	0	716821	0	1558705

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	4289	1478	108009	0	0	7	0	116783
13:FROM-GM-ONLY	7598	3974	129438	1	0	65	0	141076
10:MCIOC-SOUT	4962	0	21512	4	0	3836	0	30314
07:MCIOC-ONLY	63873	0	37728	14	0	240893	0	342508
10:CCIOC-SOUT	39467	53756	28416	1	0	3049	0	124689
07:CCIOC-ONLY	140694	65726	49873	6	0	228066	0	484365
05:SOUT-ONLY	146095	0	0	0	0	0	0	146095
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	55300	0	22752	13	0	240905	0	318970
05:FLUSH-BACK	12864	0	10809	0	0	65	0	23738
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	475142	127934	408537	39	0	716886	0	1728538

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	55757	58214	1404117	0	0	91	0	1518179
13:FROM-GM-ONLY	98774	51662	1682694	13	0	845	0	1833988
10:MCIOC-SOUT	49620	0	215120	40	0	38360	0	303140
07:MCIOC-ONLY	447111	0	264096	98	0	1686251	0	2397556
10:CCIOC-SOUT	394670	537560	284160	10	0	30490	0	1246890
07:CCIOC-ONLY	984858	460082	349111	42	0	1596462	0	3390555
05:SOUT-ONLY	730475	0	0	0	0	0	0	730475
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	110600	0	45504	26	0	481810	0	637940
05:FLUSH-BACK	64320	0	54045	0	0	325	0	118690
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	2936185	1107518	4298847	229	0	3834634	0	12177413

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	413421	3613425	13945318	110342	18082506	691572	18774078
W	77340	4703249	312416	1853	5094866	155438	5250304
DW	0	0	0	0	0	255259	255259
LR	228	1521361	607	3209	1525405	360758	1886163
UW	4347	737209	15	870	742441	31967	774408
U	4088	1109029	1	240	1113358	32765	1146123
TOTAL	499432	11684273	14258357	116514	26558576	1527759	28086335

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	372822	156460	529282	162290	691572
W	0	68110	68110	87328	155438
DW	0	0	0	255259	255259
LR	0	352529	352529	8229	360758
UW	0	31955	31955	12	31967
U	0	0	0	32765	32765
TOTAL	372822	609054	981876	545883	1527759

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	970384
ISSUED	255259
WITHOUT-SWAP-OUT	109164
WITH-SWAP-OUT	146095

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

109164	DW-WITHOUT SOUT
26667740	T-HIT + DW-WITHOUT-SOUT
1418595	T-MISS - DW-WITHOUT-SOUT
94.95 [%]	HIT-RATIO
5.05 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
301	2606	4002	343	0	940	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	INVALID	TOTAL
1717	2755	2757	2	0	21	0	940	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
1631	0	0	0	0	5129	0	6760

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	28086335	12177413	0.434
1	7	28086335	11919554	0.424
1	6	28086335	11661695	0.415
1	5	28086335	11403836	0.406
1	0	28086335	10698456	0.381
2	8	28086335	9048271	0.322
2	7	28086335	8790412	0.313
2	6	28086335	8532553	0.304
2	5	28086335	8274694	0.295
2	0	28086335	7335748	0.261

Semigroup KLI 8 PEs
Cache parms: c256,s4,w4,l1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
R	18394866	4793969	329745	6795	0	45303	0	23570678
W	5405	0	322031	6794	0	46672	0	380882
IW	489863	0	0	0	0	0	0	489863
LR	295475	0	0	0	0	21801	0	317276
UW	294557	0	0	0	0	21801	0	316358
U	3359	0	0	0	0	0	0	3359
TOTAL	19483525	4793969	651756	13589	0	135577	0	25078416

TABLE ISSUED-BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
F	7697	3198	6636	2173	0	26508	0	46212
FI	13619	0	6855	185	0	26051	0	44710
IV	7147	0	4164	1349	0	25399	0	38059
TOTAL	26463	3198	17655	3707	0	77958	0	128981

TABLE BUS USE TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	1484	347	2864	657	0	220	0	5572
13:FROM-GM-ONLY	253	286	1083	69	0	139	0	1830
10:MCTOC-SOUT	2281	0	2358	652	0	2871	0	8162
07:MCTOC-ONLY	5290	0	2613	764	0	22905	0	31572
10:CCTOC-SOUT	1130	1695	2144	31	0	922	0	8922
07:CCTOC-ONLY	5878	870	2429	185	0	25502	0	34864
05:SOUT-ONLY	98567	0	0	0	0	0	0	98567
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV ONLY	7147	0	4164	1349	0	25399	0	38059
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	125030	3198	17655	3707	0	77958	0	227548

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	19292	4511	37232	8541	0	2860	0	72436
13:FROM-GM-ONLY	3289	3718	14079	897	0	1807	0	23790
10:MCTOC-SOUT	22810	0	23580	6520	0	28710	0	81620
07:MCTOC-ONLY	47030	0	18291	5348	0	160335	0	221004
10:CCTOC-SOUT	41300	16950	21440	310	0	9220	0	89220
07:CCTOC-ONLY	41146	6090	17003	1295	0	178514	0	244048
05:SOUT-ONLY	492835	0	0	0	0	0	0	492835
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	14294	0	8328	2698	0	50798	0	76118
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	571996	31269	139953	25609	0	432244	0	1301071

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	27354	243579	21252856	677	23524466	46212	23570678
W	1657	599353	34698	487	736195	11895	748090
IW	0	0	0	0	0	122655	122655
LR	0	280220	778	40	289038	28238	317276
UW	992	308713	1974	82	311781	4577	316358
U	738	2453	0	0	3191	168	3359
TOTAL	30741	1542118	21290306	1286	24864671	213745	25078416

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	39734	3097	42831	3381	46212
W	0	9605	9605	2290	11895
IW	0	0	0	122655	122655
LR	0	26508	26508	1730	28238
UW	0	4576	4576	1	4577
U	0	0	0	168	168
TOTAL	39734	43786	83520	130225	213745

TABLE IW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	489863
ISSUED	122655
WITHOUT-SWAP-OUT	24088
WITH-SWAP-OUT	98567

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

24088	IW-WITHOUT-SOUT
24888759	T-HIT + IW-WITHOUT-SOUT
189657	T-MISS - IW-WITHOUT-SOUT
99.24 [%]	HIT-RATIO
0.76 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
641	6356	482	41	0	672	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	INVALID	TOTAL
6103	444	696	225	0	52	0	672	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
202	0	0	5	0	674	0	881

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	25078416	1301071	0.052
1	7	25078416	1293669	0.052
1	6	25078416	1286267	0.051
1	5	25078416	1278865	0.051
1	0	25078416	1269715	0.051
2	8	25078416	887925	0.035
2	7	25078416	880523	0.035
2	6	25078416	873121	0.035
2	5	25078416	865719	0.035
2	0	25078416	845425	0.034

Puzzle KL1 - 8 PEs
Cache parm: c256,s4,w4,l1

TABLE GIVEN CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
R	2795213	16088777	2043335	7515	0	55281	0	20990121
W	10070	0	2023447	7514	0	55324	0	2096355
DW	2724446	0	0	0	0	0	0	2724446
LR	1523988	0	0	0	0	26208	0	1550196
UW	890441	0	0	0	0	26208	0	916649
U	928792	0	0	0	0	0	0	928792
TOTAL	8872950	16088777	4066782	15029	0	163021	0	29206559

TABLE ISSUED BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
F	5974	106119	22155	2661	0	33207	0	170116
FI	141318	0	25389	313	0	26423	0	193443
IV	5612	0	12315	2332	0	32173	0	52432
TOTAL	152904	106119	59859	5306	0	91803	0	415991

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	7010	2674	18238	115	0	373	0	28410
13:FROM-GM-ONLY	2667	1656	8629	86	0	206	0	13244
10:MCTOC-SOUT	1753	0	9904	730	0	4396	0	16783
07:MCTOC-ONLY	3918	0	3463	1774	0	28021	0	37176
10:CCTOC-SOUT	44647	81057	5369	75	0	3227	0	134375
07:CCTOC-ONLY	87297	20732	1941	194	0	23407	0	133571
05:SOFT-ONLY	518433	0	0	0	0	0	0	518433
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	5612	0	12315	2332	0	32173	0	52432
05:FLUSH-BACK	29474	0	1951	22	0	71	0	31518
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	700811	106119	61810	5328	0	91874	0	965942

TABLE BUS-USE TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	91130	34762	237094	1495	0	4849	0	369330
13:FROM-GM-ONLY	34671	21528	112177	1118	0	2678	0	172172
10:MCTOC-SOUT	17530	0	99040	7300	0	43960	0	167830
07:MCTOC-ONLY	27426	0	24241	12418	0	196147	0	260232
10:CCTOC-SOUT	446470	810570	53690	750	0	32270	0	1343750
07:CCTOC-ONLY	611079	145124	13587	1358	0	163849	0	934997
05:SOUT-ONLY	2592165	0	0	0	0	0	0	2592165
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	11224	0	24630	4664	0	64346	0	104864
05:FLUSH-BACK	147370	0	9755	110	0	355	0	157590
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	3979065	1011984	574214	29213	0	508454	0	6102930

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	55708	1932709	18830576	1012	20820005	170116	20990121
W	8404	4050199	50776	536	4109915	28836	4138751
DW	0	0	0	0	0	682050	682050
LR	22549	1365195	0	3	1387747	162449	1550196
UW	772	912602	1082	35	914491	2158	916649
U	31044	865777	2370	0	899191	29601	928792
TOTAL	118477	9126482	18884804	1586	28131349	1075210	29206559

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	53959	102455	156414	13702	170116
W	0	9077	9077	19759	28836
DW	0	0	0	682050	682050
LR	0	154257	154257	8192	162449
UW	0	2157	2157	1	2158
U	0	0	0	29601	29601
TOTAL	53959	267946	321905	753305	1075210

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	2724446
ISSUED	682050
WITHOUT-SWAP-OUT	163617
WITH-SWAP-OUT	518433

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

163617	DW-WITHOUT-SOUT
28294966	T-HIT + DW-WITHOUT-SOUT
911593	T-MISS - DW-WITHOUT-SOUT
96.88 [%]	HIT-RATIO
3.12 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
72	5104	2509	48	0	459	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	INVALID	TOTAL
4901	2463	340	2	0	27	0	459	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
69	0	0	11	0	348	0	428

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	29206559	6102930	0.209
1	7	29206559	6061276	0.208
1	6	29206559	6019622	0.206
1	5	29206559	5977969	0.205
1	0	29206559	5911748	0.202
2	8	29206559	3973594	0.136
2	7	29206559	3931940	0.135
2	6	29206559	3890286	0.133
2	5	29206559	3848632	0.132
2	0	29206559	3725592	0.128

Pascal KL1 8 PEs
Cache parms: c256,s4,w4,l1

TABLE GIVEN-CPU-COMMAND(AREA)

GVNCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
R	467702	5162792	1257520	57845	0	103106	0	7048965
W	7464	0	1187777	56024	0	103236	0	1354501
DW	474336	0	0	0	0	0	0	474336
LR	446433	0	0	4649	0	45211	0	496293
UW	316979	0	0	4649	0	45211	0	366839
U	191480	0	0	0	0	0	0	191480
TOTAL	1904394	5162792	2445297	123167	0	296764	0	9932414

TABLE ISSUED BUS-COMMAND(AREA)

BUSCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
F	15111	3877	18190	12194	0	60425	0	109797
FI	26183	0	16172	1970	0	57823	0	102148
IV	13547	0	11541	10788	0	57770	0	93646
TOTAL	54841	3877	45903	24952	0	176018	0	305591

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	517	44	7793	584	0	70	0	9008
13:FROM-GM-ONLY	345	221	5510	386	0	107	0	6569
10:MCTOC-SOUT	4354	0	8181	3554	0	5268	0	21357
07:MCTOC-ONLY	10411	0	4475	7626	0	53858	0	76370
10:CCTOC-SOUT	9451	1508	5489	715	0	1009	0	18172
07:CCTOC-ONLY	16216	2104	2914	1299	0	57936	0	80469
05:SOUT-ONLY	79280	0	0	0	0	0	0	79280
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	13547	0	11541	10788	0	57770	0	93646
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	134121	3877	45903	24952	0	176018	0	384871

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	6721	572	101309	7592	0	910	0	117104
13:FROM-GM-ONLY	4485	2873	71630	5018	0	1391	0	85397
10:MCTOC-SOUT	43540	0	81810	35540	0	52680	0	213570
07:MCTOC-ONLY	72877	0	31325	53382	0	377006	0	534590
10:CCTOC-SOUT	94510	15080	54890	7150	0	10090	0	181720
07:CCTOC-ONLY	113512	14728	20398	9093	0	405552	0	563283
05:SOUT-ONLY	396400	0	0	0	0	0	0	396400
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	27094	0	23082	21576	0	115540	0	187292
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	759139	33253	384444	139351	0	963169	0	2279356

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	8618	1251937	5677521	1092	6939168	109797	7048965
W	6193	1596151	85599	606	1688549	21218	1709767
DW	0	0	0	0	0	119070	119070
LR	635	426910	342	58	427945	68348	496293
UW	399	346817	6792	249	354257	12582	366839
U	1564	178502	1950	2	182098	9382	191480
TOTAL	17409	3800397	5772204	2007	9592017	340397	9932414

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	97727	5231	102958	6819	109797
W	0	13329	13329	7889	21218
DW	0	0	0	119070	119070
LR	0	67503	67503	845	68348
UW	0	12578	12578	4	12582
U	0	0	0	9382	9382
TOTAL	97727	98641	196368	144029	340397

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	474336
ISSUED	119070
WITHOUT-SWAP-OUT	39790
WITH-SWAP-OUT	79280

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

39790	DW-WITHOUT-SOUT
9631807	T-HIT + DW-WITHOUT-SOUT
300607	T-MISS - DW-WITHOUT-SOUT
96.97 [%]	HIT-RATIO
3.03 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
13	5198	1587	176	0	1218	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	INVALID	TOTAL
4651	1375	812	70	0	66	0	1218	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
472	0	0	28	0	2450	0	2950

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	9932414	2279356	0.229
1	7	9932414	2263779	0.228
1	6	9932414	2248202	0.226
1	5	9932414	2232625	0.225
1	0	9932414	2199780	0.221
2	8	9932414	1617048	0.163
2	7	9932414	1602271	0.161
2	6	9932414	1586694	0.160
2	5	9932414	1571117	0.158
2	0	9932414	1520256	0.153

AQueen : KL1 - 8 PEs
Cache param: c256,s4,w4,t1

TABLE GIVEN CPU COMMAND(AREA)

CPUCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
R	1986451	10382701	1316393	105243	0	200262	0	13991050
W	7523	0	1167492	94189	0	200298	0	1469502
DW	548048	0	0	0	0	0	0	548048
LR	319698	0	0	0	0	81609	0	401307
UW	291648	0	0	0	0	81609	0	373257
U	76807	0	0	0	0	0	0	76807
TOTAL	3230175	10382701	2483885	199432	0	563778	0	16859971

TABLE ISSUED-BUS COMMAND(AREA)

BUSCMD	HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
F	35361	761	25054	32367	0	118014	0	211557
FI	43235	0	17950	944	0	125532	0	187661
IV	34396	0	9270	24618	0	113285	0	181569
TOTAL	112992	761	52274	57929	0	356831	0	580787

TABLE BUS-USE-TYPE(OPERATION)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	1235	100	10109	508	0	141	0	12093
13:FROM-GM-ONLY	1155	87	11055	341	0	155	0	12793
10:MCTOC-SOUT	16151	0	7495	8545	0	9252	0	41443
07:MCTOC-ONLY	18714	0	4070	22416	0	104166	0	149366
10:CCTOC-SOUT	10920	234	6521	171	0	1254	0	19100
07:CCTOC-ONLY	30421	340	3754	1330	0	128578	0	164423
05:SOUT-ONLY	78223	0	0	0	0	0	0	78223
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV ONLY	34396	0	9270	24618	0	113285	0	181569
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	191215	761	52274	57929	0	356831	0	659010

TABLE BUS-USE-TYPE(CYCLE)

CYCLE:PATTERN	HEAP	INST	GOAL	SUSP	META	COMM	ETC	TOTAL
13:FROM-GM-SOUT	16055	1300	131417	6604	0	1833	0	157209
13:FROM-GM-ONLY	15015	1131	143715	4433	0	2015	0	166309
10:MCTOC-SOUT	161510	0	74950	85450	0	92520	0	414430
07:MCTOC-ONLY	130998	0	28490	156912	0	729162	0	1045562
10:CCTOC-SOUT	109200	2340	65210	1710	0	12540	0	191000
07:CCTOC-ONLY	212947	2380	26278	9310	0	900046	0	1150961
05:SOUT-ONLY	391115	0	0	0	0	0	0	391115
05:SOUT-EXTRA	0	0	0	0	0	0	0	0
02:INV-ONLY	68792	0	18540	49236	0	226570	0	363138
05:FLUSH-BACK	0	0	0	0	0	0	0	0
05:FLUSH-EXTRA	0	0	0	0	0	0	0	0
TOTAL	1105632	7151	488600	313655	0	1964686	0	3879724

TABLE PREVIOUS-STATE(AREA) ALL.: ALL-AREA

CPUCMD	EC	EM	SC	SM	T-HIT	T-MISS	TOTAL
R	167300	1135912	12473629	2652	13779493	211557	13991050
W	7984	1686105	151299	1724	1847112	33021	1880133
DW	0	0	0	0	0	137417	137417
LR	137	282483	0	206	282826	118481	401307
UW	1262	307496	27725	615	337098	36159	373257
U	1381	63089	0	1	64471	12336	76807
TOTAL	178064	3475085	12652653	5198	16311000	548971	16859971

TABLE MISS-ANALYSIS(AREA) ALL.: ALL-AREA

CPUCMD	FRMC	FRCC	FRCACHE	FROM-GM	T-MISS
R	190809	8595	199404	12153	211557
W	0	22617	22617	10404	33021
DW	0	0	0	137417	137417
LR	0	116162	116162	2319	118481
UW	0	36149	36149	10	36159
U	0	0	0	12336	12336
TOTAL	190809	183523	374332	174639	548971

TABLE DW(DIRECT-WRITE)-ANALYSIS(AREA) ALL.: ALL-AREA

GIVEN	548048
ISSUED	137417
WITHOUT-SWAP-OUT	59194
WITH-SWAP-OUT	78223

TABLE CACHE-HIT-RATIO(AREA) ALL.: ALL-AREA

59194	DW-WITHOUT-SOUT
16370194	T-HIT + DW-WITHOUT-SOUT
489777	T-MISS - DW-WITHOUT-SOUT
97.10 [%]	HIT-RATIO
2.90 [%]	MISS-RATIO

TABLE CACHE-DIRECTORY-STATE Snapshot-after-execution

EC	EM	SC	SM	C	I	UNUSED	TOTAL
45	6168	452	355	0	1172	0	8192

TABLE CACHE-DIRECTORY-AREA Snapshot-after-execution

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	INVALID	TOTAL
5059	166	1753	3	0	39	0	1172	8192

TABLE BUSCMD-IS-CHANGED-BECAUSE-OF-BUS-COLLISION

HEAP	INST	GOAL	SUSP	META	COMM	ETC.	TOTAL
1044	0	0	113	0	7922	0	9079

TABLE BUS-TRAFFIC-RATIO

BUS-WIDTH[W]	MEM-ACC-TIME	MEM-REF	BUS-CYCLE	TRAFFIC-RATIO
1	8	16859971	3879724	0.230
1	7	16859971	3854838	0.229
1	6	16859971	3829952	0.227
1	5	16859971	3805066	0.226
1	0	16859971	3741101	0.222
2	8	16859971	2803756	0.166
2	7	16859971	2778870	0.165
2	6	16859971	2753984	0.163
2	5	16859971	2729098	0.162
2	0	16859971	2640947	0.157

References

[1] *Quintus Prolog User's Guide and Reference Manual Version 6*, April 1986.

[2] P. Bitar and A. M. Despain. Multiprocessor Cache Synchronization. In *13th Annual International Symposium on Computer Architecture*, pages 424-433. Tokyo, IEEE Computer Society, June 1986.

[3] P. Borgwardt and D. Rea. Distributed Semi-Intelligent Backtracking for a Stack-Based AND-Parallel Prolog. In *Symposium on Logic Programming*, pages 211-222. IEEE Computer Society, 1986.

[4] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[5] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Ltd., Wokingham, England, 1986.

[6] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, IL 60439, 1986.

[7] R. Butler et. al. Scheduling OR-Parallelism: an Argonne Perspective. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565-1577. University of Washington, MIT Press, August 1988.

[8] A. Calderwood. Scheduling Or-Parallelism in Aurora the Manchester Scheduler, July 1988. submitted for publication.

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

[10] M. Carlsson, K. Danhof, and R. Overbeek. A Simplified Approach to the Implementation of AND-Parallelism in an OR-Parallel Environment. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565-1577. University of Washington, MIT Press, August 1988.

[11] J. Chassin, J. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *Proceedings of ECAI*, pages 278-283, August 1988.

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

[13] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *Journal of the ACM*, 8:1–49, January 1986.

[14] K. L. Clark and S. Gregory. Notes on the Implementation of PARLOG. *Journal of Logic Programming*, 2(1). April 1985.

[15] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, New York. 1981.

[16] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Symposium on Logic Programming*, pages 457–467. San Francisco, IEEE Computer Society, August 1987.

[17] J. S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[18] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[19] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In *Fourth International Conference on Logic Programming*, pages 576–600. University of Melbourne, MIT Press, Cambridge MA, May 1987.

[20] B. S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[21] I. Foster and S. Taylor. Flat PARLOG: A Basis for Comparison. Research Report DOC 87/5, Imperial College of Science and Technology, March 1987.

[22] K. Furukawa, A. Okumura, and M. Murakami. Unfolding Rules for GHC Programs. *New Generation Computing*, 6(2,3):143–157, 1988. Also available as ICOT TR-277, and in France-Japan AI&CS Symposium, 1987.

[23] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge MA, 1985. Also available from Stanford University Computer Science Dept. as Research Paper 111.

[24] A. Goto, 1987. Internal ICOT Memo.

[25] A. Goto. Parallel Inference Machine Research in FGCS Project. In *Proceedings of the First Japan-U.S. AI Symposium*, pages 21–36, December 1987.

[26] A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy Reference Counting. In *Fifth International Conference and Symposium on Logic Programming*, pages 1241–1256. University of Washington, MIT Press, August 1988.

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

[28] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[29] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Fourth International Conference on Logic Programming*, pages 257–275. University of Melbourne, MIT Press, Cambridge MA, May 1987.

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

[31] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[32] E. Lusk et. al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, 1987.

[33] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, Tokyo, November 1988.

[34] D. Maier and D. S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA 94025, 1988.

[35] A. Matsumoto et. al. Locally Parallel Cache Design Based on KL1 Memory Access Characteristics. Technical Report 327, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, 1987.

[36] G. J. Myers. *Advances in Computer Architecture, Second Edition*. John Wiley and Sons, 1982.

[37] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, North Holland, May 1988.

[38] K. Nishida et. al. Evaluation of the Effect of Incremental Garbage Collection by MRB on FGHC Parallel Execution Performance. In *COMPCON Fall 88*, San Francisco, 1988. IEEE Computer Society. submitted for publication.

[39] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Symposium on Logic Programming*, pages 224–233. San Francisco, IEEE Computer Society, August 1987.

[40] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk. Prolog on Multiprocessors. Internal report, Argonne National Laboratory, Argonne, IL 60439, 1985.

[41] P. Van Roy. A Prolog Compiler for the PLM. Master's thesis, University of California at Berkeley, August 1984. Also available as Technical Report UCB/CSD 84/203.

[42] D. E. Sanger. I.B.M. Signals Big Shift in Designing Computers. *New York Times*, December 24 1987.

[43] M. Sato, 1988. personal communication.

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

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

[46] K. Seo and T. Yokota. Pegasus: A RISC Processor for High-Performance Execution of Prolog Programs. In C. H. Sequin, editor, *VLSI '87*, pages 261–274. IFIP, North-Holland, Amsterdam, 1988.

[47] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[48] E. Shapiro, editor. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge MA, 1987.

[49] K. Shen and D.H.D. Warren. A Simulation Study of the Argonne Model for OR-Parallel Execution of Prolog. In *Symposium on Logic Programming*, pages 54–68. San Francisco, IEEE Computer Society, August 1987.

[50] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge MA, 1986.

[51] P. Szeredi. More Benchmarks of Aurora. unpublished, Manchester University, March 1988.

[52] S. Takagi. A Collection of KL1 Programs Part I. Technical Memo TM-311, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, May 1987.

[53] K. Taki. Measurements and Evaluation for the Multi-PSI/V1 System. In *France-Japan Artificial Intelligence and Computer Science Symposium*, pages 359–384, Cannes, November 1987.

[54] E. Tick. Lisp and Prolog Memory Performance. Technical Report CSL-TR-86-291, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, January 1986.

[55] E. Tick. A Prolog Emulator. Technical Note CSL-TN-87-321, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, May 1987.

[56] E. Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[57] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[58] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.

[59] K. Ueda. Making Exhaustive Search Programs Deterministic: Part II. In *Fourth International Conference on Logic Programming*, pages 356–375, University of Melbourne, MIT Press, Cambridge MA, May 1987.

[60] D. H. D. Warren. Logic for Compiler Writing. *Software Practice and Experience*, 10:97–125, 1980.

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

[62] D. H. D. Warren. Prolog Engine. Technical report, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, April 1983. Unpublished draft.

[63] D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science, Springer-Verlag, March 1987.

[64] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog – Abstract Design and Implementation. In *Symposium on Logic Programming*, pages 92–102, San Francisco, IEEE Computer Society, August 1987.

[65] D. H. D. Warren and F. C. N. Pereira. An Efficient, Easily Adaptable System For Interpreting Natural Language Queries. Research Paper 155, Dept. of Artificial Intelligence, University of Edinburgh, February 1981.

[66] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symposium on Logic Programming*, pages 436-448, San Francisco, IEEE Computer Society, August 1987.