Memory Referencing Characteristics and Caching Performance of AND-parallel Prolog on Shared-Memory Multiprocessors

M. Hermenegildo

MCC

E. Tick

University of Oregon

Abstract This paper presents the performance analysis results for the RAP-WAM AND-Parallel Prolog architecture on shared-memory multiprocessor organizations. The goal of this parallel model is to provide inference speeds beyond those attainable in sequential systems, while supporting conventional logic programming semantics. Special emphasis is placed on sequential performance, storage efficiency, and low control overhead. First, the concepts and techniques used in the parallel execution model are described, along with the general methodology, benchmarks, and simulation tools used for its evaluation. Results are given both at the memory reference level and at the memory organization level. A two-level shared-memory architecture model is presented together with an analysis of various solutions to the cache coherency problem. Finally, RAP-WAM shared-memory simulation results are presented. It is argued that the RAP-WAM model can exploit coherent caches and attain speeds in excess of 2 MLIPS with current shared-memory multiprocessing technology for real applications that exhibit medium degrees of parallelism.

§1 Introduction

Logic programs²⁵⁾ offer many opportunities for parallelism¹⁰⁾. The RAP-WAM execution model^{21,22)} is aimed at providing inference speeds to logic programs beyond those attainable in sequential systems through the use of parallelism, while supporting the conventional semantics of logic languages. In particular, the model supports parallel execution of Prolog programs by exploiting Restricted/Independent AND-parallelism. The approach taken in this model is that of combining parallelism and advanced compiler technology. Therefore, in RAP-WAM the techniques used for supporting parallel execution

are efficient extensions of those used in the Warren Abstract Machine (WAM)⁴¹⁾, which have already brought high inferencing speeds to sequential implementations. Special attention is given to the preservation of WAM sequential performance and storage efficiency, and to the use of low overhead mechanisms for controlling parallel execution.

This paper offers an overview of the RAP-WAM model and the type of parallelism exploited in it, and presents simulation results supporting the claims of performance, low overhead, and storage efficiency. High-performance processing elements (PEs) are often limited in execution speed by available memory bandwidth. The bandwidth requirement is an even more critical factor in parallel systems. For this reason, the simulations were performed at the memory reference level. Because the RAP-WAM model (as the WAM) is specified at a level above that of memory organization, simulations were first performed under the ideal assumption of a uniform, single shared-memory with no contention, and some results are presented at this level. Practical memory organizations deviate from this ideal behavior, however, and it is thus important to assess the effect of this deviation if realistic speedup figures are to be obtained. The results of such a study represent the core of this paper. A special emphasis is put on the issue of cache coherency and results from the simulation of two such protocols are presented. These results are of special importance because of the unique referencing characteristics that Prolog exhibits on cache memories³⁵⁾.

The memory organization studied is characteristic of many new-generation high-performance shared-memory multiprocessors. The results obtained are thus helpful in estimation of the performance of AND-parallel Prolog/RAP-WAM on such machines. The results can be used to determine the advantages and shortcomings of the parallel implementation of other don't-know non-deterministic logic programming languages. In addition, the memory model and simulation results can also be used as an aid in the design of special-purpose multiprocessors.

Organization of the paper is as follows: Section 2 introduces the types of parallelism currently exploited in the RAP-WAM model and Section 3 gives an overall description of the RAP-WAM architecture. Section 4 summarizes the results obtained from high-level simulations of the architecture. Section 5 then presents the two-level shared-memory organization model, introduces the problems associated with cache coherency maintenance in such a model, and proposes solutions for them. Simulation results for the different coherency protocols proposed are then presented and discussed.

§2 Goal Independence and Restricted AND-parallelism

Of the various sources of parallelism present in Logic Programs¹¹⁾ AND- and OR-parallelism (or their combination) offer special promise and are currently being considered in several parallel logic programming systems^{7,34,6,4,11,45,5,27,15)}. Efficient techniques for implementing OR-parallelism

have been proposed and are currently under development by various groups, as reported by Warren^{42,43}. 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 parallel logic programming systems which exploit this type of parallelism do not implement the conventional “don’t know” non-deterministic semantics of logic programs²⁵. Instead, these systems implement *committed-choice* (i.e., “don’t care”) non-determinism^{20,33,37}. Our interest, however, is in directly supporting the conventional (“don’t know”) semantics. Although the use of program transformations has been suggested for implementing a form of “don’t know” non-determinism with committed-choice languages^{9,38,39} it is argued that a more direct approach (such as the RAP-WAM model considered in this paper or the models described by Warren⁴²) has a higher performance potential while guaranteeing that full “don’t know” non-determinism will be supported*¹.

An alternative way of dealing with variable binding conflicts which makes support for both AND-parallelism and “don’t know” non-determinism amenable to an efficient implementation is that of *Independent AND-parallelism*. In this form of AND-parallelism, only sets of goals which are determined to be independent (i.e., which do not share any non-ground variables) can be executed in parallel. Goal independence can be *postulated* through annotations³⁰. Goal independence can also be *determined* statically by the compiler (guided perhaps by some information provided by the user as in Chang⁶), or completely dynamically at run-time as in Conery¹¹, or through a combination of the above mentioned techniques.

Restricted AND-parallelism, as presented in DeGroot’s seminal work^{13,14}, represents such a combination offering more opportunity for parallelism than static approaches at a lower cost than dynamic ones. Also, this model restricts the types of execution graphs that can be exploited in exchange for implementation efficiency. The RAP-WAM model²¹ uses a generalization of these ideas²⁴*² which completes DeGroot’s original description (by providing backward execution semantics). The RAP-WAM model also introduces Conditional Graph Expressions (CGEs) that overcome the difficulty (within DeGroot’s framework) in expressing ‘sufficient’ conditions for independence²¹*³. In this approach, the

- * 1 It might be noted that, in the opinion of the authors, such programs are also easier to debug than their committed choice translations.
- * 2 The model is also currently being extended to support OR-parallelism—using techniques similar to those proposed by other researchers, see for example^{43,45} and their references—and a form of dependent AND-parallelism.
- * 3 CGEs offer Prolog syntax and permit conjunctive checks, thus lifting limitations in the expressions proposed by DeGroot: given “f(X, Y, Z):- g(X, Y), h(Y, Z).” the most natural annotation for this clause, that g and h can run in parallel if the terms in X and Z don’t share variables and Y is bound to a ground term, can be expressed easily with CGEs (“f(X, Y, Z):- (indep(X, Z), ground(Y) | | g(X, Y) & h(Y, Z).”) but is cumbersome with DeGroot’s expressions.

parallelism itself can be generated by the user by including annotations (CGEs) in the program, or automatically by the compiler, which performs a global (abstract interpretation-based) analysis⁴⁴⁾ of the program that often makes run-time independence checks unnecessary. In addition, module entry mode annotations are permitted that are also used to reduce or eliminate run-time tests and guide the global analysis. As an example consider the following clause from the *deriv* (symbolic derivation) benchmark⁴⁰⁾:

$\text{deriv}(U+V, X, DU+DV):- \text{deriv}(U, X, DU), \text{deriv}(V, X, DV).$

Without any further information from the user or any knowledge about the rest of the program, and if forced to work only locally (at the clause level) the RAP-WAM compiler performs a worst case analysis by simply applying the “independence conditions”²¹⁾. Such analysis determines that $\text{deriv}(U, X, DU)$ and $\text{deriv}(V, X, DV)$ will be independent (and therefore executable in parallel) provided that *at run-time* two conditions are met. First, the variable X must be instantiated to a ground term. Second, the variables in the two goals (U, V , and DU, DV) must not “share” variables, i.e., they contain only terms that are pairwise independent²⁴⁾. The result of this compile-time analysis can be encoded in a CGE and the *deriv* clause rewritten as shown below.

$\text{deriv}(U+V, X, DU+DV):- (\text{ground}(X),$
 $\text{indep}([U,V],[U,DV],[DU,V],[DU,DV])) \mid \mid$
 $\text{deriv}(U, X, DU) \ \& \ \text{deriv}(V, X, DV)).$

If at run-time the *ground* and *indep* checks fail, the goals in the body of the clause are executed sequentially. The backward execution semantics (actions taken in case of failure) supports “don’t know” non-determinism and is given by Hermenegildo²⁴⁾. *ground* and *indep* tests can be expensive operations and it is therefore desirable to keep them to a minimum. As mentioned above this is done through a global analysis of the program and/or user annotation. For example, a typical user annotation for *deriv* may postulate that it will always be called with the first two arguments ground. The checks are then immediately reduced to

$\text{deriv}(U+V, X, DU+DV):- (\text{indep}(DU, DV) \mid \mid$
 $\text{deriv}(U, X, DU) \ \& \ \text{deriv}(V, X, DV)).$

which is often a simple address comparison. Such annotations are, however, seldom necessary: as shown in Reference 44 the compiler can determine input and output modes through abstract interpretation-based global analysis typically for 70-80% of the arguments. For example, given a typical usage of *deriv* within a larger program the compiler can determine not only the groundness information but also that the third argument will be a variable, so that all checks are eliminated:

$$\text{deriv}(U+V, X, DU+DV):- (\text{true} \mid \mid \text{deriv}(U, X, DU) \& \text{deriv}(V, X, DV)).$$

§3 The RAP-WAM Architecture

The RAP-WAM architecture is an extension of the Warren Abstract Machine (WAM) architecture⁴¹⁾ capable of executing logic programs in parallel as determined by CGE's. A fundamental design objective of RAP-WAM is fast sequential execution for cases where there is no available (AND) parallelism. To this end, the CGE semantics has been integrated naturally into the WAM storage model in the form of specialized stack frames and storage areas that are used during parallel execution. Thus the default (sequential) model is that of a standard WAM exhibiting the same high sequential performance. Special emphasis has also been given to efficiency in the management of parallelism so that most of the WAM performance and storage optimizations are still supported during parallel execution. Figure 1 shows one of the abstract machines of the RAP-WAM architecture mapped onto a single processing element (PE). Each of these abstract machines is similar to a standard WAM (with a complete set of registers and data areas, called a *Stack Set*), with the addition of a *Goal Stack*, a *Message Buffer*, and two new types of stack frames: *Parcall Frames* and *Markers*. During an inference, goals that are ready to be executed in parallel are pushed by the abstract machine onto its own Goal Stack. From there the goals can be picked-up by this or other abstract machines for execution. Each entry in the Goal Stack is called a *Goal Frame*. Parcall Frames are used for coordinating and synchronizing the parallel execution of the goals inside a parallel call, both during forward execution and during backtracking. Markers are used to delimit *Stack Sections* (horizontal cuts through the Stack Set of a given abstract machine, corresponding to the execution of different goals) and they implement the storage recovery mechanisms during backtracking²³⁾ in a similar manner to choice-points in the WAM. In practice, the stack is divided into separate *Control* stacks (Choice Point and Markers) and *Local* stacks (Environments) for reasons of locality and locking. Table 1 summarizes the types of objects allocated in these areas and their locality.

The instruction set of the RAP-WAM architecture includes all WAM instructions and several additional instructions related to parallel execution. Table 2 lists the additional instructions that currently support AND-parallelism and the WAM instructions that are modified in RAP-WAM. Although the check instructions are somewhat particular to the implementation of RAP-WAM, the instruction set is also suitable for other AND-parallel systems. Control instructions are responsible for scheduling and synchronization of parallel goals. Control instructions of the form *..._det_...* are optimized for determinate execution and they provide significant performance advantage when no backtracking is needed. Instructions like *pop_foreign_goal*, *idle*, *redo*, etc.

Fig. 1 Data Areas and Registers for One RAP-WAM Abstract Machine

Table 1 Characteristics of RAP-WAM Storage Objects

Frame Type	Location	In WAM ?	Need lock ?	Locality
Envts./Control	Stack	Yes	No	Local
Envts./Perm. Vars.	Stack	Yes	No	Global
Choice Points	Stack	Yes	No	Local
Heap	Heap	Yes	No	Global
Trail entries	Trail	Yes	No	Local
PDL entries	PDL	Yes	No	Local
Parcall F./Local	Stack	No	No	Local
Parcall F./Global	Stack	No	No	Global
Parcall F./Wait, Sched	Stack	No	Yes	Global
Markers	Stack	No	No	Local
Goal Frames	Goal Stack	No	Yes	Global
Messages	M. Buffer	No	Yes	Global

Table 2 Paralle Abstract Machine-Specific Instructions

Control Instructions	Control Instructions (Det. Exec.)
push_call Procedure/Arity, Slot#	push_det_call Procedure/Arity, Slot#
allocate_pcall #_of_slots, M	allocate_det_pcall #_of_slots, M
check_ready Slot_#, Label	
pop_pending_goal	pop_pending_goal
wait_on_siblings	cut_merge
Modified WAM Instructions	“Check” Instructions
proceed	check_me_else Label
fail	check_ground Vn
	check_independent Vn, Vm
Pseudo-Instructions	
idle	redo
pop_foreign_goal	unwind
kill	

are *pseudo-instructions* that represent the actions taken upon failure and during distributed scheduling and backtracking. Space limitations here make a complete description of the execution model impossible. The reader is referred to Hermenegildo²¹⁾ for a complete description of the RAP-WAM instruction set and storage model.

§4 High-Level Simulation Results

As mentioned in Section 1, high-performance processing elements (PEs) in parallel systems can be severely limited in execution speed by available memory bandwidth. This is often a result of the gap in both technology and cost between PEs and interconnection networks, in addition to physical packaging/interface limitations. Because bandwidth can play a key role in accessing system performance, the simulations presented were performed at the memory reference level. Although the evaluation of the implementation of the model could be done on an existing shared-memory machine, this approach only provides a single data point corresponding to a particular organization. Also, some statistics are difficult to gather from an actual implementation. However, simulation, the approach taken in this evaluation, provides data over a wide range of

Fig. 2 Simulation tools

architectural and organizational parameters. The RAP-WAM architecture (as is its relative, the WAM) is a model described at a level above *that of memory organization design*. A series of measurement tools have been built in order to evaluate the potential performance of the execution model and the associated architectural tradeoffs at this level²¹). The present configuration of these tools is shown in Figure 2. Application programs are translated into parallel RAP-WAM code and then fed into the RAP-WAM emulator together with emulation parameters such as the number of PEs, scheduling strategy used, and storage area sizes. The emulator then executes the program generating instrumentation data such as instruction and pseudo-instruction frequencies, number of references to the different data areas, ratios of local vs. remote references for each type of object, maximum amount of storage used in each area, estimated timings, and speedups. At this level the measurements are made under the assumption of an idealized memory model with a uniform address space and no contention. The measurements are thereby made independent of the particular architectural organization on which the model is implemented. In this section results are presented that correspond to the execution of the following set of benchmarks:

- Symbolic derivation (**deriv**): this is a version of the symbolic derivation program in Warren's thesis⁴⁰). It finds the symbolic derivative of a given arithmetic expression.
- Takeuchi (**tak**): this program computes Takeuchi's function as described in Gabriel¹⁶).
- Quick-sort (**qsort**): the quick-sort algorithm written using difference lists

Table 3 Statistics for the Benchmarks (on 8 Processors)

Parameter	deriv	tak	qsort	matrix
Instructions executed	33520	75254	237884	95349
Failures	2010	1186	10368	0
Unifications	1232	3560	27804	258
Bindings	784	7117	26203	8321
References (-locks)	85477	178967	502717	96013
References (WAM)	82519	169599	499526	95357
Overhead (refs.)	2958	10268	3191	656
Overhead (%)	3.58	6.05	0.63	0.68
Goals in parallel	97	263	97	24
Idle-time (%)	13.58	7.79	29.60	9.72

as given in Clocksin and Mellish⁸⁾.

- **Matrix multiplication (matrix):** a naive matrix multiplication program where matrices are represented as lists of lists.

Each benchmark was executed on relatively large input data. Table 3 shows some statistics regarding the benchmarks used, running on eight PEs. Some observations regarding this table are appropriate: the table lists the number of instructions, failures, unifications, variable bindings, and memory references for each benchmark. The number of references is seen to be sufficient for vigorously exercising the memory simulators that will be described in the next section. For comparison, the number of references generated by a sequential WAM running the benchmarks using the Van-Roy compiler³¹⁾ is also listed. The difference between these two numbers represents the overhead incurred due to control of parallelism (for eight PEs). The overhead running sequentially on one RAP-WAM PE is typically 1-3%, i.e., only 1-3% more references are generated by the parallel versions of the benchmarks running on one RAP-WAM PE than by the sequential versions running on WAM. This low overhead is attained by supporting *all* the WAM performance optimizations within RAP-WAM sequential and parallel execution: customization of unification, clause indexing, hashing on arguments, efficient management of backtracking, storage recovery on backtracking and of local storage, etc.

The number of references listed in Table 3 does not include references generated by spin locks, since these references are expected to be absorbed by the cacheing mechanism, as confirmed in the results presented in the next sections. The number of goals actually executed in parallel and the percentage of time that processors are idle (because there is no work available in the system) are also listed. These benchmarks, and the input data used, were chosen for several reasons: first, because of their small granularity (except for matrix) their execution offers a worst-case type of analysis with respect to parallelism management overhead. They also offer reasonable degrees of parallelism so that the parallel portion of the abstract machine is exercised. Finally, as will be shown in Section 5.2, the benchmarks' sequential memory referencing behavior and characteristics resemble those of much larger Prolog programs, such as the ones studied by Tick³⁵⁾.

Table 4 Referencing Characteristics, **deriv** on 8 Processors

deriv												
Frame	Reads	%	rem	%	Writes	%	rem	%	R + W	%	rem	%
choice-p	19062	20.8	0	0.0	19566	21.3	0	0.0	38628	42.1	0	0.0
env-cont	5778	6.3	0	0.0	12475	13.6	0	0.0	18253	19.9	0	0.0
env-pvar	4748	5.2	0	0.0	3391	3.7	0	0.0	8139	8.9	0	0.0
heap	5724	6.2	5230	5.7	10182	11.1	97	0.1	15906	17.3	5327	5.8
trail	0	0.0	0	0.0	1554	1.7	0	0.0	1554	1.7	0	0.0
pf-contr	262	0.3	0	0.0	337	0.4	97	0.1	599	0.6	97	0.1
pf-locks	5697	6.2	291	0.3	558	0.6	291	0.3	6255	6.8	582	0.6
markers	194	0.2	0	0.0	776	0.8	0	0.0	970	1.1	0	0.0
goal stk	714	0.8	679	0.7	714	0.8	0	0.0	1428	1.6	679	0.7
<i>Totals</i>	42179	46.0	6200	6.8	49553	54.0	485	0.5	91732	100	6685	7.3
<i>Pot. rem</i>	16985	18.5	6200	6.8	14942	16.3	485	0.5	31927	34.8	6685	7.3

Table 4 offers a summary of the memory referencing behavior of RAP-WAM running the **deriv** benchmark on eight PEs. Rows classify references according to the area of the stack set and type of frame being accessed. These include choice points (in the control stack), environments (in the local stack), heap objects (in the heap), trail entries (in the trail), parcall frames (in the local stack), markers (in the control stack), and goal frames (in the goal stack). Areas not referenced in this benchmark (such as the message buffer, only used during deep, non-deterministic backtracking between parallel goals) are not listed. References to environments are further divided into references to permanent variables, and references to the control parts of the environment. References to parcall frames are also further divided into references to the control part of the frame and references to the locks used for synchronization. The *Totals* line adds up references from the different storage areas.

Columns in Table 4 classify references into reads and writes, and totals (reads + writes). Within each of these categories, four numbers are listed: the number of references in absolute and the percentage they represent of the total number of references, and the number of these references that are *remote* (i.e., access the address space—stack set—of another PE) and their percentage. The last row represents the number of references that could not be determined statically (at compile-time) to be local and had to be labeled as “potentially remote”. This labeling is used in the hybrid cache coherency algorithms studied in the next sections.

It is interesting to note that 65% of the total number of references could be determined to be local at compile-time. However, of the remaining 35% potentially remote references, only 7.3% actually ended up accessing data within other PEs’ spaces. Note also that most remote references correspond to accesses to the heap and are mostly reads, due to structure copying. The referencing characteristics at this level reflect those observed by Tick³⁵⁾ for large Prolog programs during sequential WAM execution. In fact, RAP-WAM inherits both advantageous and disadvantageous characteristics, such as the larger than normal proportion of writes to reads (around 1:1) and the large number of

Table 5 Referencing Characteristics (Summary), on 8 Preprocessors

Frame	Reads	%	rem	%	Writes	%	rem	%	R+W	%	rem	%
"tak"												
Totals	70790	39.1	3948	2.2	110422	60.9	1315	0.7	181212	100.0	5263	2.9
Pot. rem	27638	15.3	3948	2.2	23177	12.8	1315	0.7	50815	28.0	5263	2.9
"qsort"												
Totals	210110	41.7	11862	2.4	293480	58.3	485	0.1	503590	100.0	12347	2.5
Pot. rem	66750	13.3	11862	2.4	91134	18.1	485	0.1	157884	31.4	12347	2.5
"matrix"												
Totals	50060	52.0	16608	17.3	46143	48.0	120	0.1	96203	100.0	16728	17.4
Pot. rem	37587	39.1	16608	17.3	29444	30.6	120	0.1	67031	69.7	16728	17.4

references to choice points. However, the same compilation techniques that hold promise for alleviating these defects in the WAM (e.g.,^{28,36,2,32}) can similarly be used with RAP-WAM.

Table 5 offers a summary of the data contained in Table 4 for the other benchmarks further supporting the observations made above. Note the unusually large number of potentially shared (70%) and actually shared (17.4%) references in **matrix**. This is due to the extensive use and sharing of the heap, because of the implementation of the arrays as lists of lists. Other conclusions regarding the RAP-WAM model based on these and other results from the memory reference level simulations performed are summarized as follows:

- Again, the overhead in the RAP-WAM model due to the management of parallelism is low. For example, overhead has been observed to be less than 15% for up to 40 processors even for fine granularity (i.e., high overhead) cases such as that of the **deriv** benchmark, as shown in Figure 3. In this figure, *work time* represents the number of cycles spent by all PEs doing actual processing (i.e., not *waiting* or *idle*). Overhead, the

Fig. 3 RAP-WAM Overheads for deriv

difference between the work done by RAP-WAM and the work done by WAM, is the distance between the work curve and a horizontal line corresponding to WAM work, represented by the “uniprocessor” line in Figure 3. All data in this figure is presented as percentages of WAM work (i.e., 100%=the number of cycles necessary to execute the sequential version of benchmark on the WAM). Note that the RAP-WAM work on one PE is within 2% of the work done by WAM.

Therefore, it appears that the RAP-WAM model can achieve *actual speedup* (i.e., significantly faster execution than a high-performance sequential implementation—WAM—for similar performance PEs) even if the application exhibits only low levels of parallelism. This is because the RAP-WAM offers sequential speed essentially equivalent to the WAM and low overhead in the management of parallelism. Speedup measurements are presented in Figure 4 for the **deriv** benchmark. The main limitation to speedup in this example is the parallelism available in the benchmark, dependent on input data size (c.f., the “idle” curve in Figure 3). Speedups are attainable under the assumption, of course, that the memory organization can handle the associated traffic. This issue is treated in the following sections.

The stack-based memory management approach²³⁾ appears to be very efficient. Its implementation is, on the other hand, more complicated than that of other solutions, such as a simple heap. However, it is well worth the effort: it extends to a parallel execution environment the storage management optimizations of the WAM, i.e., recovery of storage on backtracking, recovery of local storage upon exit from a procedure, last call optimization, etc. This means that the storage requirements for the RAP-WAM system running in parallel are essentially the same as those

Fig. 4 RAP-WAM Speedups for deriv

of the WAM, and that, as in the WAM, the invocation of garbage collection is minimized.

Although these results are encouraging, further evaluation taking into consideration the characteristics of a particular memory organization (such as available bandwidths, cache coherence maintenance overhead, etc.) on the performance of the model is necessary in order to guarantee actual speedup. This subject is addressed in the next sections.

§5 The Two-Level Shared-Memory Architecture Model

In contrast with the idealized memory organization used in the high-level simulations presented previously, practical shared-memory systems generally present a two-level structure where a local cache memory is located between each PE and the system bus, as shown in Figure 5. Such a hierarchical organization serves two main purposes. First, it allows faster execution because of the generally lower effective memory access time seen by a PE, in much the same way as in a sequential system. Second, it absorbs a (hopefully) significant part of the traffic to main memory that needs to go through the system bus. In shared-memory multiprocessors the latter is particularly important because the system bus is often the most significant bottleneck in the system. The former is essential in obtaining performance that is competitive with that of sequential systems. The locality of Prolog/WAM was shown by Tick³⁵. In the next sections it is shown that Prolog/RAP-WAM also offers sufficient locality to take advantage of cache memories.

Fig. 5 The Two-Level Shared-Memory Architecture Model

5.1 Cache Coherency

Many of the local memory designs used in conventional or special-purpose sequential machines for the implementation of logic programs (e.g., PSI-II²⁹) cannot be used directly in a shared-memory Multiprocessor because of *coherency* problems^{*1}. *Coherent caches* ensure that all the PEs in the system have a consistent view of the storage model. Recall that the RAP-WAM architecture ensures that processes running in parallel will not attempt to make conflicting bindings. Thus cache coherency during process execution is not *required*: however, prior to a process spawn or upon process completion, cache coherency *is* required at least for the correct management of process control structures, such as the Parcall Frames. It appears, however, that ensuring coherency *continually* is easier than enforcing coherency only at specific points (and has the additional benefit of generality). Therefore, traditional coherent caches are considered in this study.

Historically, the first coherent caches, proposed by Gibson¹⁷, used a *write-through strategy*, where all write references were issued to both the local cache and shared memory. In addition, remote copies of a write target, i.e., copies residing on a cache other than the cache issuing the write request, were invalidated. This *coherency protocol* is inexpensive in terms of hardware, but offers low performance because of excessive traffic on the system bus.

Recently, a family of *fully distributed broadcast cache protocols* have been proposed and built^{18,1,33}. Broadcast protocols are based on the ability of the cache organization to modify all copies of a cached item in all caches which share this item *in a single bus cycle*. Information is maintained for each cache block as to whether it is *private* or *shared*, making it possible to avoid coherency overheads for private blocks. In addition, it is also possible to issue write references to the local cache only (and the caches which also contain this item) and *not* to shared memory. Shared memory may be independently updated during replacement of a modified cache block (write-back), for instance, and need not be consistent at all times with the local caches. The broadcasting mechanism guarantees, however, that local caches will still be consistent at all times among themselves. Different designs differ essentially in the treatment of a write to a possibly shared block. A write-through broadcast strategy *updates* remote copies and possibly shared memory. A write-in broadcast strategy *invalidates* remote copies. Descriptions and measurements of the relative performance of various broadcast protocol attributes for conventional architectures are given in Archibald¹¹.

The broadcast protocol offers high performance at the expense of additional hardware. With the objective of reducing this expense by exploiting attributes of the RAP-WAM architecture, a (*firmware*) *controlled hybrid cache*

*1 Note that simple buffers which hold only local data, such as a choice point buffer, do not suffer from this coherency problem and can be included without modification in a parallel system.

protocol was developed. This scheme attempts to combine the efficiency of broadcast caches with the simplicity and low cost of a traditional write-through cache using information provided by the PE (in the form of tags, derived from the information in Table 1) as to the locality characteristics of each reference. The protocol is referred to as “hybrid” because based on these tags *potentially shared* (global) data is written-through and local data is copied back*¹. An underlying tenet of the hybrid protocol is to avoid some of the complexity of broadcast caches by keeping shared memory consistent with local memory. The cost associated with this simplification is the traffic required to write-through to memory the write requests marked as global that are not actually shared. The gain with respect to the traditional write-through approach is that data marked as local is not written-through.

5.2 Simulations

In order to compare the performance of the various types of caches presented above, the RAP-WAM emulator was modified to generate a trace file of memory references. These references are marked with a *PE identifier*, a tag describing the particular storage area and object being accessed, and a read/write flag. All of the coherent cache models are simulated with the same parameterized multiprocessor cache simulator³⁵⁾ which can be reconfigured to support the various consistency protocols. The simulator takes the memory trace file as input (see Figure 2) and processes trace records sequentially, using the cache corresponding to the record's PE identifier. Each private cache contains a number of blocks of a given size (in words). Caches are modeled as fully associative memories with perfect LRU replacement. Cache consistency is maintained for each reference. The simulator models a system with no cache-to-cache transfer capability. Therefore in the broadcast models, if the most up-to-date version of a miss target is held by a remote cache, the line is first copied back to memory, and then transferred to the requesting cache. Note that this is a pessimistic assumption since many systems can perform these two actions simultaneously. On the other hand, the simulator makes the optimistic assumption that no unnecessary invalidations/write-throughs are generated for lines that are assumed to be shared but all remote copies have been replaced.

The results presented in this section were obtained with execution traces for the four benchmarks listed in Section 4. Each benchmark was executed on large input data in order to generate address traces of sufficient size to fully exercise all caches in the multiprocessor model (a “cold start” was measured). It is hypothesized that the conclusions drawn from this study can be extrapolated to RAP-WAM Prolog programs in general. At the very least, the experiments allow comparisons between alternative coherent memory designs. Evidence supporting the above hypothesis is offered below. First, because the benchmarks

*1 A similar capability, of both write-through and copy-back, is supported by the Fairchild Clipper microprocessor¹²⁾.

Table 6 Fit of Small Benchmarks to Large Benchmarks (Traffic Ratio)

cache size (words)	E_{tr}	σ_{tr}	$(tr-E_{tr})/\sigma_{tr}$				
	large bench		deriv	tak	qsort	(matrix)	mean
512	0.164	0.0626	1.1	-1.9	0.83	(16.0)	1.3
1024	0.108	0.0569	2.0	-1.1	1.6	(5.2)	1.6

Fig. 6 Traffic of Coherency Schemes

measured have small granularity, the study represents a worst case analysis of parallelism management overhead. Second, the benchmarks' sequential memory referencing behavior resembles that of much larger Prolog programs³⁵). Third, the *locality* characteristics of the benchmarks also resemble those of the larger Prolog benchmarks. Table 6 gives the mean and standard deviation of the traffic ratios of large WAM benchmarks for four-word-block copyback data caches³⁵). Also shown are the errors, in number of standard deviations, of the small benchmarks studied here. The fit is quite good, except for **matrix**, which performs poorly on a uniprocessor (Prolog has no arrays), but which performs quite well on a multiprocessor because of data partitioning. The average does not include **matrix**. This fit ensures that the benchmarks exercise the sequential storage model (the foundation of the RAP-WAM storage model) in a reasonable, typical way.

Figure 6 shows the mean traffic ratios (averaged over the four benchmarks) of the write-in broadcast, hybrid, and conventional write-through cache protocols, using four word lines. Caches of sizes 64, 128, and 256 words were

Fig. 7 Traffic of Benchmarks for 8 PE's

simulated with no-write-allocate (a write miss does not fetch the corresponding block to cache). Caches of sizes 512 and 1024 words were simulated with write-allocate, except for hybrid caches which used no-write-allocate for 512 words. These selections were made on the basis of the policy which produced the lowest traffic. A clear result of the simulations is that no-write-allocate is best for small caches; however, miss ratio increases with no-write-allocate. Another result is that a more efficient replacement policy (e.g., copyback) produced lower traffic with write-allocate than a less efficient policy (e.g., hybrid) for the same cache size.

The write-through broadcast cache statistics (not shown in Figure 6) are almost identical to those of the write-in broadcast cache, an indication that communication traffic in RAP-WAM is low. For each type of cache a family of curves is represented, corresponding to different numbers of PEs. Each curve plots traffic ratio as a function of *total* local memory size (i.e., the sum of the individual PE cache sizes).

A result seen from the curves is that the hybrid cache does quite well in reducing traffic, almost to the level of the copyback cache. The copyback cache does exceedingly well for 1024 word caches, and this trend is expected to continue with larger sizes, because the hybrid caches have already bottomed-out.

The idiosyncrasies in the curves are due to the effects of averaging the benchmarks. In addition, the advantageous effect (that of reducing memory traffic) of partitioning an algorithm's working set across several caches is seen to sometimes outweigh the increase in communication overheads. Figure 7 shows separate curves for each of the benchmarks for hybrid and copy-back coherent caches (for eight PEs).

5.3 Discussion

The hierarchical memory organization serves the dual purpose of lowering the effective memory access time and reducing the memory bandwidth requirement of a PE. According to the results of the simulations presented in the previous section, the hybrid cache generates an amount of traffic between that generated by the broadcast and conventional write-through caches. The broadcast schemes retain a (sometimes slight) advantage throughout the range of caches simulated.

It should be noted that these results measure performance only in terms of traffic ratio. For example, the simulation data shows that eight PEs with write-in broadcast caches (of 128 words or greater) generate a traffic ratio of less than 0.3 (the hybrid cache is also close to this performance). This means that more than 70% of the traffic generated by the processors is captured in the local memories and will not appear on the bus. In order to accurately estimate the actual performance of a multiprocessor the time penalty to access shared memory due to contention must also be taken into account. This analysis can be done statistically with a queueing model. Although beyond the scope of this

paper such a model for RAP-WAM execution is presented in Tick³⁵⁾.

It is of obvious interest, if only to stimulate further research, to speculate about the potential performance levels attainable given the results presented in the previous sections. Even current low- to medium-cost shared-memory systems offer high PE to memory bandwidths by implementing multiple or overlapped busses and interleaved memories. This makes it reasonable to predict that speeds in the order of two million application*¹ inferences per second are possible on shared-memory multiprocessors built using current technology*². A “back of the envelope” calculation, in order to justify this claim and based on the results obtained from the present and previous studies can be made as follows. Studies of large Prolog benchmarks show that in the average 15 (WAM or RAP-WAM) instructions are executed per logical inference (LI) and that each instruction averages 3 (word) references. This represents 45 words/LI, or 180 bytes/LI for a 32 bit word size. Therefore, a system executing at a speed of 2 MLIPS would require a cumulative memory bandwidth of 360 Mbytes/sec. If the caches are able to capture 70% of this traffic, about 110 Mbytes/sec have to be delivered by the bus/memory system, a performance which is achievable with current technology (and expected of the upcoming generation of shared-memory multiprocessors)*³.

§6 Conclusions

The paper presents memory referencing characteristics of a parallel logic programming architecture, RAP-WAM, based on Independent/Restricted AND-parallel execution of Prolog. We describe its behavior and potential performance first for an idealized memory model and then for shared-memory multiprocessor organizations. The goal of the RAP-WAM model is to provide inferencing speeds to logic programs beyond those attainable in sequential systems through the use of parallelism, while preserving the conventional semantics of logic languages. The main objectives in developing such a multiprocessor architecture are to ensure high performance single PE execution, storage efficiency, and minimal communication and synchronization overheads.

The measurements presented here indicate that RAP-WAM is well-suited for high performance execution on tightly-coupled shared-memory multiprocessors, from cost-effective small-scale systems to higher performance medium-sized systems. It is argued that speeds of two million application inferences per second are possible with currently available technology for applications which exhibit medium degrees of parallelism. It has been shown that the architecture offers

- *1 Application inferences refer to inference steps of the average size found in large Prolog programs, i.e., on the order of 15 WAM instructions³⁵⁾. This results in much lower but more realistic figures than those obtained using the conventional “LIPS” measurement based on “naive reverse.”
- *2 Note that the Japanese FGCS Project is also predicting similar inferencing speeds for the PIM¹⁹⁾.
- *3 These conclusions are in disagreement with a related study by Fagin¹⁵⁾ and his contention that Prolog programs cannot effectively make use of multiprocessing. Our results do agree, however, with those of Lin²⁶⁾.

high memory referencing locality so that it can take advantage of two-level memory organizations. The memory referencing characteristics presented included comparison of cache coherency protocols, indicating how well alternative designs perform under the expected communication loads. The "broadcast" and "hybrid" protocols are shown to offer superior performance to write-through mechanisms, present in some multiprocessors.

Because the memory organizations studied are characteristic of many current and next-generation multiprocessors, it is argued that the results obtained are relevant to the estimation of the performance of AND-parallel Prolog/RAP-WAM on these multiprocessors. The results also help determine the advantages and shortcomings of the parallel implementation of other don't-know non-deterministic logic programming languages. In addition, the memory model and simulation results can also be used as an aid in the design of small to medium-sized special-purpose multiprocessors. Although the goal of small to medium systems may seem rather unambitious, it is important to have evidence of actual speedups at these levels before attempting the design of large-scale systems.

§7 Acknowledgements

The authors would like to thank Richard Warren for his comments on earlier drafts of this paper and the Microelectronics and Computer Technology Corporation for its support. Evan Tick was affiliated with Stanford University during this research and supported by an IBM Graduate Fellowship and by NASA-Ames under consortium NCA2-1R745-406.

References

- 1) Archibald, J., High Performance Cache Coherence Protocols For Shared-Bus Multiprocessors. Technical Report 86-06-02, University of Washington, Seattle, WA 98195, June 1986.
- 2) Asakawa, Y., Komatsu, H., Kurokawa, T. and Tamura, N., A Very Fast Prolog Compiler on Multiple Architectures. In *Fall Joint Computer Conference*. ACM and IEEE Computer Society, November 1986.
- 3) Bitar, P. and Despain, M., Multiprocessor Cache Synchronization. In *13th Annual International Symposium on Computer Architecture*, pp. 424-433. IEEE Computer Society, June 1986.
- 4) Borgwardt, P. and Rea, D., Distributed Semi-Intelligent Backtracking for a Stack-Based AND-Parallel Prolog. In *1986 International Symposium on Logic Programming*, pp. 211-222. IEEE Computer Society, 1986.
- 5) Butler, R., Lusk, E. L., Olson, R. and Overbeek, R. A., ANLWAM: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, IL 60439, 1986.
- 6) Chang, J. H., Despain, A. M. and DeGroot, D., AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Digest of Papers of COMPCON Spring '85*, pp. 218-225, 1985.
- 7) Ciepielewski, A. and Haridi, S., Control of Activities in the OR-Parallel Token

- Machine. In *1984 International Symposium on Logic Programming, Atlantic City*, pp. 49-58, Silver Spring, MD, February 1984. IEEE Computer Society Press.
- 8) Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*. Springer-Verlag, Berlin, New York, 1981.
  - 9) Codish, M. and Shapiro, E., Compiling OR-Parallelism into AND-Parallelism. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 283-298. Springer-Verlag, July 1986.
  - 10) Conery, J. S., *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California at Irvine, 1983. Technical Report 204.
  - 11) Conery, J. S., *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.
  - 12) Fairchild Corp. Introduction to the CLIPPER Architecture. Fairchild Camera and Instrument Corp., Palo Alto CA 94304, 1986.
  - 13) Degroot, D., Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pp. 471-478, November 1984.
  - 14) DeGroot, D., Restricted AND-Parallelism and Side-Effects. In *Symposium on Logic Programming*, pp. 80-89. IEEE Computer Society, August 1987.
  - 15) Fagin, B. and Despain, A., Performance Studies of a Parallel Prolog Architecture. In *14th Annual International Symposium on Computer Architecture*, pp. 108-116. IEEE Computer Society, June 1987.
  - 16) Gabriel, R. P., *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge MA, 1985. Also available from Stanford University Computer Science Dept. as Research Paper 111.
  - 17) Gibson, D. H., Considerations in Block-Oriented Systems Design. In *AFIPS Conference Proceedings*, Vol. 30, pp. 75-80. Spring Joint Computer Conference, Academic Press, April 1967.
  - 18) Goodman, J. R., Using Cache Memory to Reduce Processor-Memory Traffic. In *10th Annual International Symposium on Computer Architecture*, pp. 124-131. IEEE Computer Society, 1983.
  - 19) Goto, A., Parallel Inference Machine Research in FGCS Project. In *Proceedings of the First Japan-U.S. AI Symposium*, pp. 21-36, December 1987.
  - 20) Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation*. Addison-Wesley Ltd., Wokingham, England, 1987.
  - 21) Hermenegildo, M. V., *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.
  - 22) Hermenegildo, M. V., An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 25-40. Springer-Verlag, 1986.
  - 23) Hermenegildo, M. V., Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*. University of Melbourne, MIT Press, Cambridge MA, May 1987.
  - 24) Hermenegildo, M. V. and Nasr, R. I., Efficient Management of Backtracking in AND-parallelism. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 40-55. Springer-Verlag, 1986.
  - 25) Kowalski, R. A., Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pp. 569-574, 1974.

- 26) Lin, Y.-J., *A Parallel Implementation of Logic Programs*. PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.
- 27) Lin, Y.-J., Kumar, V. and Leung, C., An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 55-69. Springer-Verlag, 1986.
- 28) Mellish, C. S., Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, Vol. 2, No. 1, April 1985.
- 29) Nakashima, H. and Nakajima, K., Hardware Architecture of the Sequential Inference Machine: PSI-II. In *1987 International Symposium on Logic Programming*, pp. 104-113. IEEE Computer Society, August 1987.
- 30) Pereira, L. M. and Nasr, R. I., Delta-Prolog: A Distributed Logic Programming Language. In *Proceedings of the Intl. Conf. on 5th. Gen. Computer Systems*, 1984. Japan.
- 31) Van Roy, P., 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.
- 32) Van Roy, P. and Demoen, B., Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. Technical report, University of Leuven, August 1986.
- 33) Shapiro, E. Y., A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, JAPAN, January 1983.
- 34) Shen, K., An Investigation of the Argonne Model of OR-Parallel Prolog. Master's thesis, The University of Manchester, November 1986.
- 35) Tick, E., *Studies In Prolog Architectures*. PhD thesis, Stanford University, Stanford, CA 94305, June 1987.
- 36) Turk, A. K., Compiler Optimizations for the WAM. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 657-662. Springer-Verlag, 1986.
- 37) Ueda, K., *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.
- 38) Ueda, K., Making Exhaustive Search Programs Deterministic. In *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pp. 270-283. Springer-Verlag, 1986.
- 39) Ueda, K., Making Exhaustive Search Programs Deterministic: Part II. In *Fourth International Conference on Logic Programming*, pp. 356-375. University of Melbourne, MIT Press, Cambridge MA, 1987.
- 40) Warren, D. H. D., *Applied Logic—Its Use and Implementation as Programming Tool*. PhD thesis, University of Edinburgh, 1977. Also available as SRI Technical Note 290.
- 41) Warren, D. H. D., An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.
- 42) Warren, D. H. D., OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.
- 43) Warren, D. H. D., The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Symposium on Logic Programming*, pp. 92-102. IEEE Computer Society, August 1987.
- 44) Warren, R., Hermenegildo, M. and Debray, S., On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, August 1988.
- 45) Westphal, H. and Robert, P., The PEPsSys Model: Combining Backtracking, AND- and OR-Parallelism. In *Symposium of Logic Programming*, pp. 436-448. IEEE Computer Society, August 1987.