Stack-splitting: A Simple Technique for Implementing Or-parallelism and And-parallelism on Distributed Machines[1]

Gopal Gupta and Enrico Pontelli

Laboratory for Logic, Databases and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
`http://www.cs.nmsu.edu/lldap`

And then Krishna held up a twig, pointed it towards Bhima, and split it through the middle.

– The Mahabharata

Abstract:

We study the problem of exploiting *or-parallelism* from logic programming systems on distributed machines. We propose *stack-splitting*, a modification of the well-known *stack-copying* technique for environment representation, that can be used for exploiting or-parallelism efficiently from logic programming systems on distributed-memory machines. Stack-splitting coupled with *scheduling on bottom-most choice-point* leads to: (i) reduced communication during distributed execution; and, (ii) distribution of larger grain-sized work to processors. The modified technique can also be implemented on shared memory machines and performs better than standard stack-copying implementations such as Muse. We also show how stack-splitting can be adapted for realizing distributed memory implementations of and-parallelism.

1 Introduction

An important property of *logic programming (LP)* languages is that they are single assignment languages. Unlike conventional programming languages they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed (to a certain extent) in any order without affecting the (declarative) meaning of the program. In particular, these operations can be performed *in parallel*. An important characteristic of LP languages, thus, is that parallelism is implicitly present in the operational model of the language and can be (partially or completely) exploited in an *implicit* way. This can be done directly by the program evaluator—i.e., the runtime system—as suggested above, or, alternatively, it can be done by a parallelizing compiler. The task of the parallelizing compiler is essentially to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user, through suitable source-level annotations. In all cases, the advantage offered by LP is that the process is easier because of the declarative, high level nature of the language. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model. Two principal kinds of (implicitly exploitable) parallelism can be identified in LP:

1. *Or-parallelism (OP)* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other, giving rise to OP. OP is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel.
2. *And-parallelism (AP)* arises when more than one goal is present in the query or in the body of a clause, and these goals are concurrently reduced by parallel threads. Thus, AP is a way of efficiently developing a specific solution to the query, by constructing different parts of the solution in parallel. Typically two forms of AP are distinguished. *Independent and-parallelism* arises when more than one goal is present in the body of a clause, and the runtime bindings for the variables in these goals are such that two or more

[1] This work has been partially supported by NSF grants CCR 96-25358, HRD 96-28450, INT 95-15256, CDA 97-29848, CDA 98-02251, and a grant from the US-Spain Fullbright Foundation program.

goals are *independent* of one another, i.e., their resulting argument terms after applying the bindings of the variables are either *ground* or have non-intersecting sets of variables.[2] Parallel execution of such goals gives rise to independent AP. *Dependent and-parallelism* arises when two or more goals in the body of a clause have a common variable and are executed in parallel.

Research on parallel LP has been going on for almost 20 years. Significant progress has been made and many implementation techniques devised that have led to efficient parallel systems [27, 24, 1, 18, 20, 17, 16]. Most research so far has focused on techniques aimed at shared-memory multiprocessors (SMMs). Relatively fewer efforts [11, 2, 8, 7, 6] have been devoted to implementing LP systems on distributed memory multiprocessors (DMMs). Out of these efforts only a very small number have been implemented as working prototypes, and even fewer have produced acceptable speedups. One can conclude that the state-of-the-art of parallel LP systems running on DMMs is far behind that of parallel LP systems running on SMMs.

Existing techniques developed for SMMs are inadequate for the needs of DMMs. This is because most of the implementation methods need to share both the data and control stacks to work correctly. Even if the need to share data stacks is eliminated (e.g., in the *stack copying method* for OP) the need to share the control stack still exists. The control stack can be quite large, thus degrading the performance on DMMs. In this paper, we present a simple modification of the stack-copying method, called *stack-splitting*, designed to support OP on DMMs. Stack-splitting allows one to avoid sharing of the control stack. Thus, stack-splitting allows efficient techniques devised for OP on SMMs to be effectively implemented also on DMMs. Stack-splitting naturally incorporates efficient scheduling and reduced communication between processors. Stack-splitting has the potential to

- improve locality of computation, reduce communication between parallel threads, and increase memory access efficiency (e.g., better cache behavior).
- allow better scheduling strategies—specifically scheduling on bottom-most choice-point—to be realized even in distributed memory implementations of OP.

The fact that our techniques for implementing LP systems on distributed architectures are based on simple modifications of efficient techniques that have proved to work on SMMs gives us confidence that they will work better than many approaches previously proposed. Our prototype implementation (on SMMs) as well as simulation studies [22] further bolster our confidence.

Stack-splitting can also be adapted for efficient implementation of AP on DMMs, and applied to other types of LP systems, such as constraint logic programming systems. In addition, it can be efficiently implemented on shared-memory as well as distributed memory architectures. On SMMs we anticipate that the method's performance will be comparable to that of regular stack-copying [1]—our prototype implementation shows that our method's performance is, in fact, better. The real benefit of the method will be apparent on DMMs, on which efficient implementation becomes possible. The ability to reuse the same technology on both SMMs and DMMs is also a key component to allow the development of LP systems on *Clusters of SMMs*, i.e., distributed architectures whose individual nodes are shared memory machines. Developing uniform programming models for such architectures has been recognized as a challenging problem [4].

2 The Environment Representation Problem

A major problem in implementing OP is that multiple branches of the search tree are active simultaneously, each of which may produce a solution or may fail. Each of these branches may potentially create a binding for a variable created earlier during the execution—i.e., before the choice-point. In normal sequential execution, where only one branch of the search tree is active at any given time, the binding for the variable created by that branch can be stored directly in the memory location allocated for such variable. During backtracking, this binding is removed (untrail operation), so as to free the memory location for use by the next branch.

However, in OP execution, this memory location will have to be turned into a *set of locations*, shared between processors, or some other mean devised to store the multiple bindings that may *simultaneously* exist. In addition, we should be able to efficiently distinguish the bindings applicable to each branch during variable access in each branch. This problem of maintaining multiple bindings efficiently is called the *multiple environments representation* problem. An extensive discussion can be found in [14] and a complexity-theoretic

[2] More refined notions of independence have been considered as well, such as non-strict independence.

analysis of the problem is presented in [26]. Numerous solutions have been devised to solve the multiple environments representation problem, and a survey of these techniques can be found in [12].

2.1 Stack-copying for Maintaining Multiple Environments

Stack-copying [1] has been one of the most successful approaches for solving the multiple environments representation problem. It has been incorporated in the Muse OP system [1] and used as basis for more complex parallel systems (e.g., ACE [24] and Penny [20]). In this approach, processors working in OP maintain a *separate* but *identical* address space, i.e., they allocate their data areas starting at the same logical addresses. Whenever a processor $\mathcal{A}$ working in OP becomes idle (let us call $\mathcal{A}$ the *idle-agent*), it will start looking for unexplored alternatives generated by another processor $\mathcal{B}$ (let us call $\mathcal{B}$ the *active-agent*). Once a choice-point p with unexplored alternatives is detected in the computation tree $\mathcal{T}_\mathcal{B}$ generated by $\mathcal{B}$, $\mathcal{A}$ will create a local copy of $\mathcal{T}_\mathcal{B}$—from the root to the choice-point p—and restart the computation by backtracking over p and executing one of its unexplored alternatives. The fact that all the or-agents maintain an identical logical address space reduces the creation of a local copy of $\mathcal{T}_\mathcal{B}$ to a simple memory copying operation (Fig. 1), without the need of any pointer-relocation operations.

In practice, the stack-copying operation is slightly more involved than simply copying data structures, as the choice-points have to be copied to an area accessible to all processors. This is important because the set of untried alternatives is now shared between the two processors; if this set is not accessed in a mutually exclusive manner then two processors may execute the same alternative. Thus, after copying, each choice-point in $\mathcal{T}_\mathcal{B}$ will be transferred to a shared area. Using Muse's terminology, we will refer to a choice-point transferred to the shared memory area as a *shared frame*. Both the active and idle agents will replace their choice-points with pointers to the corresponding shared frames.[3] Shared frames should be accessed in a mutually exclusive manner—i.e., at most one processor at a time is allowed to access a shared frame. This whole operation of obtaining work from another processor is usually termed *sharing of or-parallel work*.

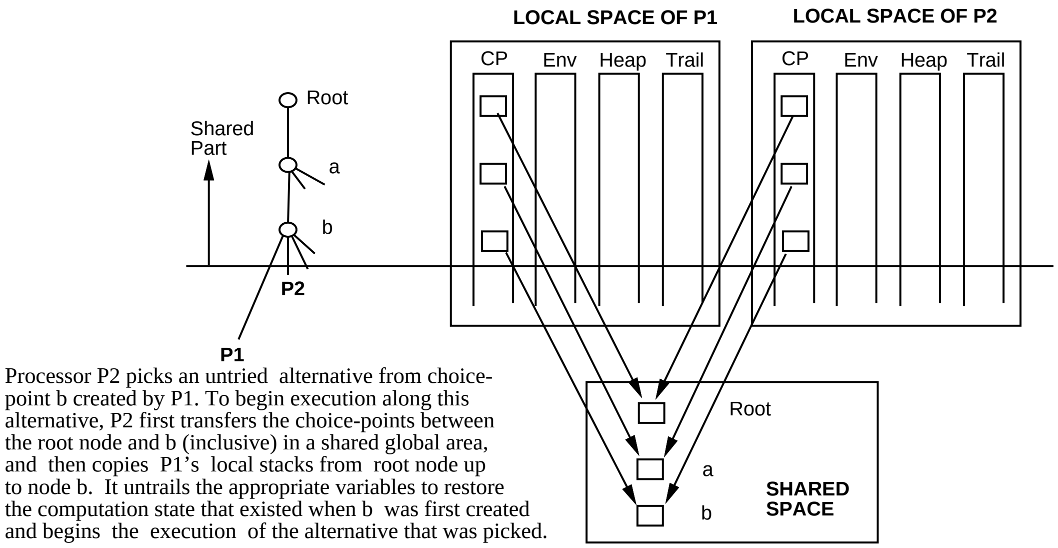

Figure 1: Stack-copying based Or-parallelism

In order to reduce the number of sharing operations performed—since each sharing operation may involve a considerable amount of overhead—unexplored alternatives are always picked from the bottom-most choice-point in the tree [1, 5]. During the sharing operation all the choice-points between the bottom-most choice-point and the topmost choice-point are shared between the two processors. This means that, in each sharing operation, we try to maximize the amount of work shared between the two processors. Furthermore, in order to reduce the amount of information transferred during the sharing operation, copying is done *incrementally*, i.e., only the *difference* between $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_\mathcal{B}$ is actually copied.

[3]Technically, only a part of the choice-point will be transferred in the shared area, e.g., the set of remaining alternatives, while the remaining part will be duplicated in the stack of each processor.

Indeed, a major reason for the success of the Muse method is that it does *scheduling on bottom-most choice-point*. That is, an idle processor picks work from the bottom-most choice-point of an or-branch. The stack segments upwards of this choice-point are copied before the exploration of this alternative is begun. The copied stack segments may contain other choice-points with untried alternatives. These alternatives will be tried via backtracking on the copied segments (of course, they may also be picked by other processors looking for work). Thus, a significant amount of work becomes available to the copying processor every time a copying operation is performed. The cost of having to copy potentially larger fragments of the computation tree is irrelevant considering that this technique drastically reduces the *number of sharing operations* performed. It is also important to observe that each sharing operation requires both the processors involved to stop the regular computation and cooperate in the sharing. Scheduling on bottom-most choice-point strategy was so successful in Muse that it was adopted in other competing or-parallel systems, e.g., Aurora [5].

3 The Split Choice-point Stack-Copying Model

Whenever backtracking encounters a shared choice-point, most information needed has to be retrieved by acquiring exclusive access to the corresponding shared frame, and then reading the information. This solution based on shared frames works fine on SMMs—where mutual exclusion is easily implemented with the aid of *locks*. However, on a DMM this process is a source of significant overhead, as the operation of accessing the shared area becomes a bottleneck [3]. This is because sharing of information in a distributed memory machine leads to frequent exchange of messages and hence considerable overhead. Centralized data structures, such as the shared frames, are, not unexpectedly, expensive to realize in a distributed setting. Nevertheless, stack copying has been considered by most authors as the best environment representation methodology to support OP in a distributed memory setting [8, 11, 2, 7, 6]. This is because, while the choice-points are shared, at least other data-structures, such as the environment, the trail, and the heap are not. Other environment representation schemes, e.g., the Binding Arrays scheme [29], have been specifically designed for SMMs and share most of the computation; the communication overhead produced by these alternative schemes on DMMs is likely to be prohibitive. To avoid the problem of sharing choice-points in distributed implementations, many implementors have reverted back to the *scheduling on top-most choice-point* strategy [8, 11]. The reasoning, essentially, is that untried alternatives of a choice-point created higher up in the or-parallel tree are more likely to have large subtrees rooted at it. Thus, if a processor steals an alternative from a choice-point higher up, the likelihood of obtaining work with large grain is considerably higher. However, if the granularity does not turn out to be large, then another untried alternative has to be picked and a new copying operation performed. In contrast, in scheduling on bottom-most more work could be found by simple backtracking, since many more choice-points are copied during the same sharing operation. Additionally, scheduling on bottom-most is closer to the depth-first search strategy used by sequential logic programming systems. Thus, if one wishes to support sequential Prolog semantics in a parallel system, then better efficiency will result with the scheduling on bottom-most strategy [1, 13]. Research done on comparing scheduling strategies indicates that scheduling on bottom-most is superior to scheduling on top-most [5]. This is especially true for the stack-copying technique because: (i) the number of copying operations is minimized; and, (ii) the alternatives in the extra choice-points that get copied are "cheap" sources of additional work, simply available via backtracking. However, the fact that these choice-points are shared is a major drawback for a distributed implementation of copying. So the question we wish to consider is: can we avoid this sharing of choice-points while keeping scheduling on bottom-most? The answer is affirmative, as is discussed next.

3.1 Split Choice-point Stack Copying

In the stack-copying technique, the primary reason why a choice-point has to be shared is because we want to make sure that the selection of its untried alternatives by various concurrent processors is serialized, so that no two processors can pick the same alternative. The shared frame is locked while the alternative is selected to achieve this effect. However, there are other simple ways of ensuring that no alternative is simultaneously selected by multiple processors: *the untried alternatives of a choice-point can be split between the two copies of the choice-point stack*. We call this operation *Choice-point Stack-Splitting* or simply *Stack-splitting*. This will ensure that no two processors pick the same alternative.

We can envision different schemes for splitting the set of alternatives between the two (or more) choice-points—e.g., each choice-point receives half of the alternatives, or the partitioning can be guided by additional information regarding the unexplored computation, such as granularity and likelihood of failure. In addition, the need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a choice-point, though their contents differ in terms of which unexplored alternatives they contain. All the choice-points can be evenly split in this way during the copying operation. The choice-point stack-splitting operation is illustrated in figure 2.

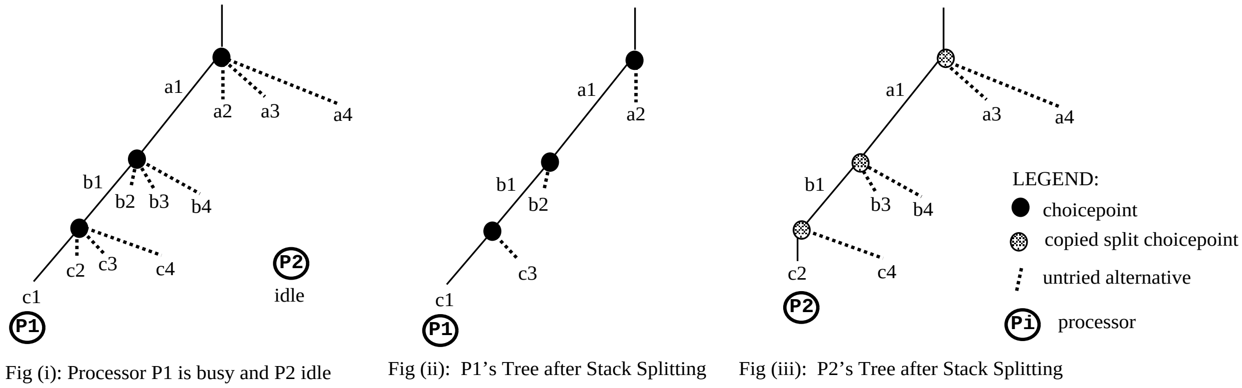

Figure 2: Stack-splitting based or-parallelism

The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting the different or-parallel threads become fairly independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for DMMs. The possibility of parameterizing the splitting of the alternatives based on additional semantic information (granularity, non-failure, user annotations) can further reduce the likelihood of additional communications due to scheduling.

3.2 Overheads of Stack-Splitting

The shared frames in the regular stack-copying technique is also a place where global information related to scheduling and work-load is kept. The shared frames provide a globally accessible description of the or-tree, and each shared frame keeps information that allows one to determine which processor is working in which part of the tree. This last piece of information is of particular importance to support the kind of scheduling typically used in stack-copying systems—work is taken from the processor that is "closer" in the computation tree, thus reducing the amount of information to be copied—since the "distance" between the processors, and the hence the difference between their stacks, is minimized. The shared nature of the frames ensures accessibility of this information to all processors, all of whom see a consistent picture of the computation.

However, under stack-splitting the shared frames no longer exist; thus, scheduling and work-load information will have to be maintained in some other way. They could be kept in a global shared area similar to the case of SMMs—e.g., by building a representation of the or-tree—or distributed over multiple processors and accessed by message passing in case of DMMs. The maintenance of global information for scheduling purposes represent a problem which is orthogonal to the environment representation. This means that scheduling management in a DMM will anyway require communication between processors.

Shared frames are also employed in Muse [1] to detect the Prolog order of choice-points, needed to execute order-sensitive builtin predicates (e.g., side-effects, extra-logical predicates) in the correct order. As in the case of scheduling, some information regarding global ordering of choice-points needs to be maintained to execute order-sensitive predicates in the correct order.

Thus, stack-splitting does not completely remove the need of a shared description of the computation tree. Nevertheless, the use of stack-splitting can mitigate the impact of accessing logically shared resources. For example, stack-splitting allows scheduling on bottom-most which, in general, leads to a reduction of the number of calls to the scheduler.

3.3 The Cost of Stack-Splitting

Let us consider the cost of the stack-splitting operation. The stack-copying operation in the stack-splitting technique is a little involved, though only slightly more, than in regular (*a lá* Muse) stack-copying. In Muse, the original choice-point stack is traversed and the choice-points transferred to the shared area. This operation involves only those choice-points that have never been shared before—if a choice-point is already shared, then its copy already resides in the global shared area. For this reason the actual *sharing* of the choice-points is performed by the *active-agent*—which is thus forced to interrupt its regular flow of computation to assist in the sharing process. The actual copying of the choice-point stack between the processors takes place only after the appropriate choice-points have been copied in the global shared memory-area.

In the stack-splitting technique, the sharing is replaced by a first phase of splitting, performed by the active agent; after the copying is done, the idle agent needs also to traverse the copied choice-points, completing the splitting of the untried alternatives. In the case of shared memory implementations, this operation is expected to be considerably cheaper than transferring the choice-point to the shared area. The actual splitting can be represented by a simple pair of indices that refer to the list of alternatives—which, in a shared memory system like Muse and ACE, is static and shared by all the processors. In the case of distributed memory implementations, the situation is similar: since each processor maintains a local copy of the code, the splitting can be performed by communicating to the copying processor which alternatives it can execute for each choice-point (e.g., described as a pair of pointers to the static list of alternatives). It is simple to encode such information within the choice-point itself at the time of copying.

Thus, in both cases we expect the sharing operation to have comparable complexity; a slight delay may occur in the case of stack-splitting, due to the need of performing a traversal of the choice-point stack in both the processors. On the other hand, in stack-splitting the two traversals—one in the *idle-agent* and one in *active-agent*—can be overlapped, while in the original stack copying scheme the *idle-agent* is necessarily suspended until the *active-agent* has completed the sharing operation. However, if the stack being copied, S_o, is itself a copy of some other stack, then unlike, regular stack-copying, we may still need to traverse both the source and the target stacks and split the choice-points. The presence of this additional step depends on the policy adopted for the partitioning of the alternatives between processors. It is, for example, required if we simply adopt a partitioning policy which assigns half of the alternatives to each of the two processors involved. In such cases, the cost of the sharing operation will be slightly more than the cost of copying in regular stack-copying on SMMs.

Once a processor selects work from another processor, it will look for work again only after it finishes the exploration of this alternative, as well as all the alternatives it acquired via stack-splitting. Incremental copying and other optimizations developed for Muse still apply to stack-splitting, though some extra work is needed. Each processor has knowledge of the part of its stack which are shared, as this information is locally available to the processor. These shared parts, if possible, should not be immediately deallocated on backtracking, as otherwise they may need to be copied again during the next sharing activity.

3.4 Supporting Prolog's Sequential Semantics

In order to support Prolog semantics, we should execute the side-effects in the order in which they would be executed in a sequential implementation of Prolog. Also, it is desirable to explore the branches in the search tree in the left to right order. Performing the side-effects in the sequential order requires considerable synchronization between the different or-branches. In a distributed implementation, this synchronization may generate considerable message traffic; however, this traffic is really independent of the environment representation technique used. As far as exploring the search space with a bias towards exploring the branches to the left is concerned, it will depend on the choice-point splitting strategy used. Consider the choice-point with alternatives `a1` through `a5` shown in Fig. 3(i). Two possible splittings are shown in Fig. 3(ii) and 3(iii). In the first one (Fig. 3(ii)), the list of alternatives is split in the middle: processor P1 will be working on the left half of the tree rooted at this choice-point `a`, processor P2 on the right half. In contrast, in Fig. 3(iii), the untried alternatives are distributed alternately between the two choice-points. This splitting strategy is more likely to produce a search that is biased to the left.

If we are interested in all solutions to a goal, then the order in which the alternatives are explored is immaterial. This is frequently the case in CLP(FD) systems, where for a forward checkable predicate, we

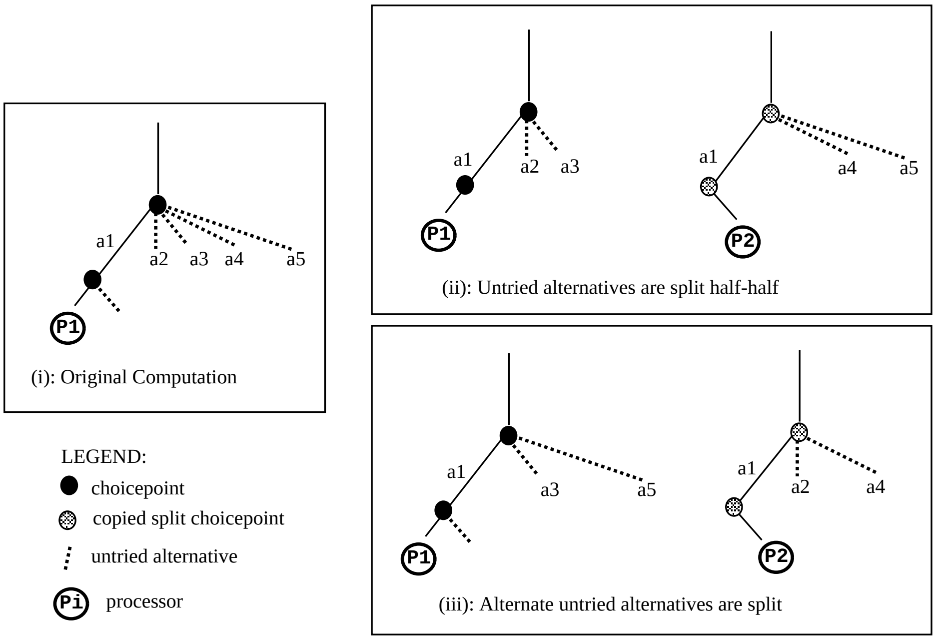

Figure 3: Distribution of Unexplored Alternatives

wish to narrow the domain to the fullest extent possible. However, if we are interested in producing the first solution as quickly as possible, then the splitting strategy used acquires considerable importance.

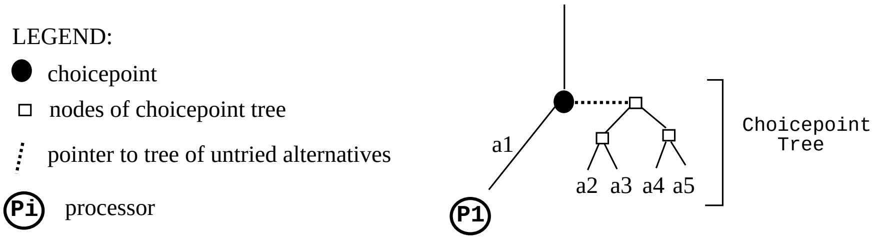

Figure 4: Amortizing Splitting Overhead

3.5 Optimizing Stack-splitting Cost

The cost incurred in splitting the untried alternatives between the copied stack and the stack from which the copy is made, can be eliminated by amortizing it over the operation of picking untried alternatives during backtracking. Let us assume that the untried alternatives will be split evenly (Fig. 3(ii)).

In the modified approach, no traversal and modification of the choice-points is done during the copying operation. The untried alternatives are organized as a binary tree (see Figure 4). Note that the binary alternatives can be efficiently maintained in an array, using standard techniques found in any data-structures textbook. In addition, each choice-point maintains the "copying distance" from the very first original choice-point as a bit string. This number is initially 0 when the computation begins. When stack-splitting takes place and a choice-point whose bit string is n is copied from, then the new choice-point's bit string is $n1$ (1 tagged to bit string n), while the old choice-point's bit string is changed to $n0$ (0 tagged to bit string n). When a processor backtracks to a choice-point, it will use its bit string to navigate in the tree of untried alternatives, and find the alternatives that it is responsible for. For example, if the bit-string of a processor is 10, then it means that all the alternatives in the left subtree of the right subtree of the binary tree are to be executed by that processor.

However, it is not very clear which of the two strategies—incurring cost of splitting at copying time *vs* amortizing the cost over the operation of picking untried alternatives—would be more efficient. In case of amortization, the cost of picking an alternative from a choice-point is, of course, now slightly higher, as the binary tree of choice-points needs to be traversed to find the right alternative.

3.6 Applicability and Effectiveness

Stack-splitting essentially performs static work distribution, as the untried alternatives are split at the time of picking work. If the choice-points that are split are balanced, then we can expect good performance. Thus, we should expect to see good performance when the choice-points generated by the computation that are parallelized contain a large number of alternatives. This is the case for applications which fetch data from databases and for most generate & test type of applications.

For choice-points with a small number of alternatives, the stack-splitting scheme is more susceptible to problems created by the static work distribution strategy that implicitly results from it: for example, in cases where OP is extracted from choice-points with only two alternatives. Such choice-points arise quite frequently, since many programs generate OP from predicates like member and select:

```
member(X,[X|_]).                    select(X,[X|Y],Y).
member(X,[_|Y]) :- member(X,Y).     select(X,[Y|Z],[Y|R]) :- select(X,Z,R).
```

Both these predicates generates choice-points with only two alternatives—thus, at the time of sharing, a single alternative is available in each choice-point. The different alternatives are spread across different choice-points. Stack-splitting would assign all the alternatives to the copying processor, thus leaving the original processor without local work. However, the problems raised by such situations can be solved using a number of techniques:

- Use knowledge about the inputs to the program for partial evaluation so as to remove occurrences of member/select, effectively collapsing at compile-time the different choice-points into a single one.
- Use automatic optimizations like the *Last Alternative Optimization* [15] which allow to recollect at run-time the different choice-points into a single one containing all the alternatives.
- Use more complex splitting strategies, for example, if a choice-point has odd number of untried alternatives remaining (say $2n + 1$), then one processor will be assigned n alternatives and the other $n + 1$. The processor which gets n and the processor which gets $n + 1$ can be alternated for the different choice-points encountered as we move up the stack.
- Perform a *vertical* splitting of the choice-points, i.e., each processor is given all the alternatives of alternate choice-points (See Figure 5). Thus, in this case, the alternatives are not split, rather the list of choice-points available is split between the two processors. In fact, vertical splitting is even more efficient than the *horizontal* splitting introduced in Section 3.1 and described thus far. Horizontal splitting can be resorted to when very few choice-points are present in the stack. Different mixes of vertical and horizontal splitting of can be tried to achieve a good balance of load. Eventually, the user can also be given control regarding how the splitting is done.

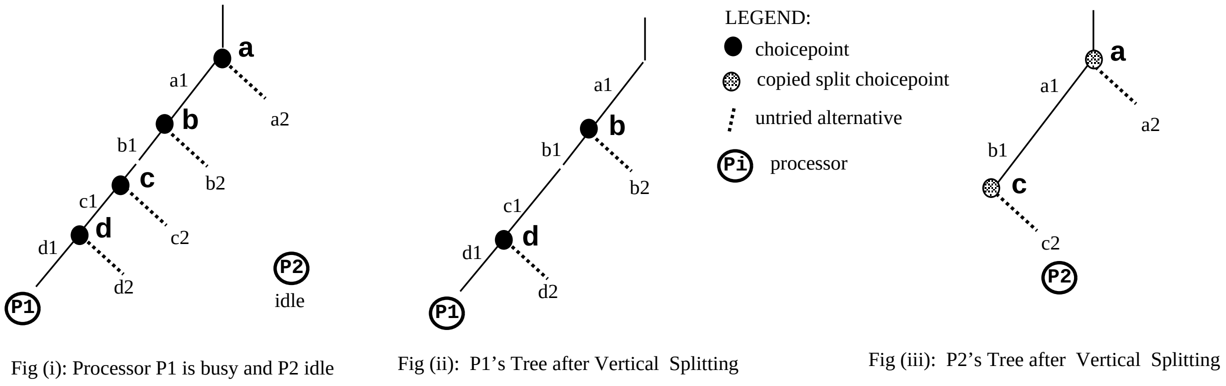

Figure 5: Vertical Splitting of Choice-points

4 Distributed Implementation of And-parallelism

We first briefly explain how and-parallel systems are traditionally implemented, then we discuss how distributed implementations of and-parallel systems can be realized using stack-splitting.

4.1 Implementation of And-parallel Systems

In traditional logic programming independent and-parallel (IAP) computations are commonly managed using the *marker based scheme* devised in the RAPWAM model [17] and refined by various researchers [21, 28, 25]. The marker model works by creating a *parcall* frame on the stack when a parallel conjunction of goals is encountered. The parcall frame keeps track of information related to execution of each of the goal in the parallel conjunction: a *goal-slot* is created in the parcall frame for every goal in the parallel conjunction; the slot keeps track of control information related to that goal. An input-marker is inserted in the stack before execution of one of the goals is begun by a processor, and an end-marker is inserted after the execution is over. The input- and end-markers are inserted to identify the portion of the stack corresponding to the goal. Backtracking becomes quite complicated in the presence of AP. Backtracking has to be done in the correct order to ensure the integrity of the execution (details can be found, for instance, in [28, 25, 21]).

4.2 Stack-splitting for Independent And-parallelism

The stack-splitting technique can also be applied to obtain a distributed implementation of AP. This observation is based on the *duality* [23] between OP and AP: a technique that benefits OP also benefits AP and vice versa. OP and AP are fundamentally the same, in that in both cases multiple threads concurrently work in the creation of a computation tree. The only difference is that while in the case of OP the threads are independent, in the case of AP these threads have to be *joined* as the effect of the computation in different threads is needed in the computation that corresponds to the continuation (the goal that comes after the parcall frame). A choice-point (which gives rise to OP) and a parcall frame (which gives rise to AP) are similar structures since both are used to manage the multiple threads that give rise to OP and AP respectively. Thus, just as choice-points can be split in our improved technique at the time a processor picks or-parallel work from another processor, parcall frames can likewise be split when and-parallel work is sought by another processor. Of course, we will have to consider how the split-computations on different processors working in and-parallel can be "joined" so that the continuation goal can be correctly executed, an issue that is absent in the case of OP. In the next subsection we present a scheme for implementing independent AP that is based on stack-splitting.

4.3 Stack-splitting based Distributed Implementation of And-parallelism

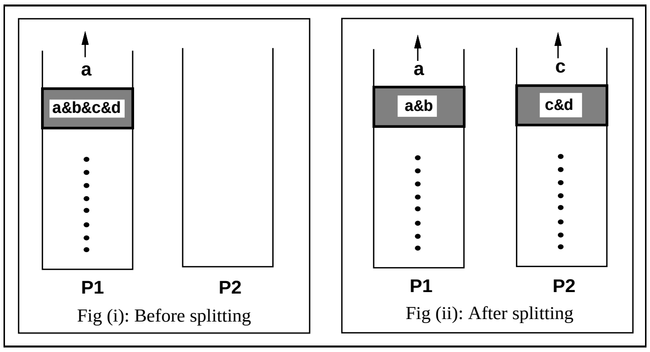

Figure 6: And-parallel stack-splitting

Consider a processor P_1 executing a goal g. Suppose after some time P_1 reaches a parallel conjunction and allocates a parcall frame. Suppose processor P_2, which is idle, requests an and-parallel subgoal from the parallel conjunction. Following the analogy with OP, the parcall frame (which is analogous to a choice-point) will be split and half of the and-parallel goals will be transferred to P_2. In addition, P_2 will copy the stacks up to and including the point where the parcall frame is allocated, so that execution of the and-parallel goals picked up by P_2 can be done independently of P_1 (See Figure 6). Copying of stack frames is done because and-parallel goals picked up by P_2 may access/bind variables that arise in the the goals occurring before the parallel conjunction. The goal-slots in the parcall frame need to be maintained as a list on the heap, so that they can be easily split. In fact, the goal-slots are maintained as a list on the heap anyway, for triggering optimizations such as the *last parallel call optimization* [16]. Most of the techniques discussed for OP (e.g., incremental stack copying, amortization of overhead of splitting, etc.) in earlier sections apply to the case of

AP, if we think of the parcall frame as analogous to a choice-point and the various and-parallel goals in the corresponding parallel conjunction as analogous to the alternatives of that choice-point. However, the main issue that we need to tackle is the joining of various and-parallel computations so that the continuation goal after the parallel conjunction can be correctly executed. This issue is addressed next.

Consider the goal `g, (p & q), r`, which indicates that the goals `p` and `q` are to be executed in independent and-parallel. During their parallel execution the goals `p` and `q` may access and/or bind variables created during the execution of the goal `g`. Likewise, the goal `r`, which is the continuation of the parallel conjunction, may access and/or bind variables created during the execution of `p`, `q` as well as `g`. The only guarantee that independence gives is that the goal `p` will never access and/or bind variables created in `q` and vice versa. The continuation goal, `r`, only needs to know the bindings created for unbound variables that are reachable from the arguments of `p` and `q`. If the unbound variables appearing in the arguments of `p` and `q` are *globalized* [12, 17], then these bindings will be found either in the stacks of the goals before the parallel conjunction or the heap (but not in the stacks of the goal corresponding to `p` and `q`). Bindings created by `p` and `q` for variables allocated space in the stacks corresponding to `g` can be easily recorded by setting up a *value-trail* in each processor. Essentially, any bindings made to a variable allocated space above the parcall frame is recorded in a separate trail (called *and-trail*). Both the address of the variable and its binding are recorded in this trail. Now the bindings created by `p` and `q` for variables that have been allocated space in `g`'s stack can be obtained by the processor executing `r` by simply copying the corresponding segment of the and-trails of processor executing `p` and `q`, and installing the bindings therein. This installation process will, also require copying terms from the heaps of `p` and `q` to the heap of the processor executing `r`, since `p` or `q` may bind a variable created in `g` to a term. Variables occuring in such a term may not be present in the and-trail. However, these terms are easily accessible as they have to be pointed by values in the and-trails of `p` and `q`. Thus, during installation, while traversing the and-trails of `p` and `q`, if a term is found, then it has to be copied onto `r`'s heap.
This process can be further optimized using various techniques, such as the copy-on-demand recently described in the DAOS system [8] and the analysis for backtracking independence [25].

4.4 Discussion

The advantage of using stack-splitting for realizing AP are similar to those for OP. When a processor steals work from another processor, it gets a large grain of work. Coupled with optimizations such as the Last Parallel Call Optimization [16] which increases the size of the parcall frame, larger grain-sized work can be obtained. Also, if additional processors are not available, optimization such as *determinate processor optimizations* [16] can keep the and-parallel computation as close to sequential execution as possible—just as in the case of OP, where a choice-point is explored via backtracking if it is not processed in or-parallel.

In our opinion, the stack-splitting technique for obtaining distributed implementations of AP should be reasonably effective. Copying of stacks may seem to be an inordinate amount of effort for and-parallel implementations, but given that implementations such as Muse are quite efficient, a stack-copying based and-parallel system should also be efficient. Also, the overhead in setting up the bindings while *joining* the multiple and-parallel threads, so that the parcall's continuation goal can be correctly executed, may be quite high; however, we hope that the use of and-trails will keep it to a minimum. It is also important to observe that the issue of joining will arise in any distributed implementation of AP, where bindings of commonly accessible variables have to be exchanged between processors to allow for a coherent execution. The use of and-trail is similar to techniques such as *environment closing* [10] that have been proposed in the past, however, we believe a combination of stack-copying and the use of and-trails is a more efficient way of realizing environment closing than the traditional one—in which the terms are explicitly traversed at the time an and-parallel goal is picked and at the time an and-parallel goal finishes execution.

Note that just as optimizations such as Last-alternative optimization lead to increased applicability of and benefit from stack-splitting in the case of OP, so will optimization such as Last-parallel call optimization lead to increased applicability of and benefit from stack-splitting in the case of AP. Finally, we only considered stack-splitting for implementing IAP, it can also be adapted for implementing DAP. Use of stack-splitting to implementing DAP is being investigated.

5 Performance Results

A prototype implementation of the stack-splitting technique has been carried out. The stack-splitting technique has been implemented by modifying the Muse or-parallel system, which is itself realized on top of the SICStus Prolog (SICStus 2.1). This prototype has been developed on SMMs. This prototype was easier to build, compared to a true distributed implementation of stack-splitting, and enabled us to (i) perform a preliminary feasibility study of our ideas, (ii) compare performance to the original stack-copying method, and, (iii) quickly obtain a system that can used for a low-level study of locality-of-reference, communication overhead, etc., using simulators: this gives us a good indication of how our technique will perform in a distributed setting. Implementation on distributed-memory architectures is currently in progress.

Benchmark	**# Processors**							
	1		**3**		**5**		**7**	
	Muse	SS	Muse	SS	Muse	SS	Muse	SS
Tina	21.4	22.5	25.1 (0.85)	10.1 (2.22)	25.1 (0.85)	8.3 (2.71)	25.0 (0.85)	7.6 (2.97)
Large	131.7	144.9	118.0 (1.12)	54.3 (2.67)	120.1 (1.1)	388.4 (3.73)	118.4 (1.11)	32.3 (4.49)
Queens1	56.8	63.6	49.8 (1.14)	39.5 (1.61)	42.9 (1.32)	36.1 (1.76)	42.1 (1.35)	34.4(1.85)
FQueens	51.3	59.0	18.8 (2.72)	24.5 (2.41)	10.8 (4.77)	13.1 (4.51)	7.5 (6.88)	9.9 (5.95)
Salt	98.8	104.7	38.7 (2.55)	36.2 (2.89)	25.1 (3.93)	23.3 (4.5)	18.6 (5.32)	18.2 (5.76)
Solitaire	22.9	24.4	8.4 (2.71)	8.7 (2.8)	5.3 (4.29)	5.6 (4.37)	4.6 (4.95)	4.1 (5.91)
Houses	37.1	41.2	13.7 (2.71)	14.5 (2.84)	8.9 (4.18)	9.3 (4.45)	6.7 (5.53)	6.9 (5.92)
constraint	67.0	69.7	25.0 (2.68)	25.7 (2.71)	15.2 (4.42)	15.5 (4.49)	10.6 (6.35)	10.8 (6.44)

Table 1: Execution Times on Sequent: Muse vs. Stack-Splitting (SS)

The timing results (in seconds) along with speedups obtained are shown in Table 1 and Table 2. All the benchmarks for which results are reported involve search. The benchmarks used are classical benchmarks used for evaluating the parallel behavior of or-parallel systems—and they have been mostly taken from the benchmarks pool of Muse and Aurora systems. The benchmarks considered are: *Tina:* is a holiday planning program from ECRC; *Large:* is a fragment of the static analyzer used in the ACE system; *Queens1:* a solution to the 8-queen problem (from ECRC); *FQueens:* another solution to the 8-queen problem; *Solitaire:* a game-playing program; *Houses:* a solution to the well known "who owns the zebra" puzzle; *Constraint:* map coloring problem using arc consistency.
The results reported in this paper have been obtained using a Pentium-Pro 200Mhz 4-node SMM (running Solaris), and an 8-node Sequent shared memory system (running Dynix). The Sequent hardware is relatively old and slow with respect to current industry standards. Nevertheless it gives a good feeling for the parallel behavior of the system and it represents the largest parallel SMM currently available to us. Experiments on Pentium-based and Sparc-based parallel systems have provided comparable speedups.

The results presented in Table 1 illustrate the execution times (in seconds) and speedups observed on the Sequent system. The figures reported are average execution times over a sequence of 10 runs. Additionally, the table presents the results obtained using the Muse or-parallel system and the or-parallel system based on Stack-splitting. Table 2 presents the results (time in milliseconds) obtained on the Pentium-pro hardware. It can be observed that the speedups on the two systems are very similar. All benchmarks have been executed by requiring the system to exploit parallelism only from selected promising predicates, and by declaring all other predicates sequential (i.e., non-parallelizable). We believe this situation better reflects the kind of behavior needed to guarantee adequate performance in a distributed execution (which remains the ultimate goal of the stack-splitting system). All the benchmarks have been executed using all-solutions queries.

As can be seen in Table 1, stack-splitting leads in general to better speed-ups. The execution time of the stack-splitting system is occasionally slightly worse than the execution times of Muse—on average the sequential stack-splitting system is 5% to 12% slower than sequential Muse. This is due to:

- the temporary removal of some sequential optimizations from the stack-splitting system, to facilitate the development of the initial prototype;
- the presence of one additional comparison during the backtracking phase—needed to distinguish between choice-points that have been split and those that have not.

The first problem will be solved in the next version of our prototype. The second problem is inherent in the current representation of the alternatives in the choice-points used by the SICStus system. All choice-points associated to the same predicate share the same list of alternatives. This complicates the implementation of stack-splitting, as the list of alternatives cannot be directly manipulated—as this may potentially affect other choice-points as well. The simple alternative of duplicating the list of alternatives proved too inefficient. At present we have introduced two pointers in the choice-point to maintain the segment of alternatives list of interest—and this leads to the need of discriminating between split and non-split choice-points during backtracking. The adoption of a different alternatives representation in the engine could solve this problem—but requires tedious changes to the compiler.

Benchmark	**# Processors**		
	1	**2**	**3**
Tina	1215	719 (1.69)	535 (2.27)
Large	8093	4422 (1.83)	3065 (2.64)
Queens1	3520	2466 (1.43)	2207 (1.59)
Salt	6117	3274 (1.87)	2155 (2.84)
Solitaire	1364	704 (1.94)	488 (2.79)
Houses	2425	1263 (1.92)	859 (2.82)
constraint	3030	1569 (1.93)	1102 (2.75)

Table 2: Execution Times on Solaris X86 for Stack-Splitting

In the case of *Tina* benchmark, the Muse system suffers a slowdown irrespective of the number of processors employed. This behavior is rather unusual (and at odds with the speed-ups reported for *Tina* in the literature for Muse). We conjecture it originates from the fact that we have explicitly identified all choice-points as parallel or sequential. Stack-splitting is instead capable of extracting parallelism from this benchmark reaching a maximum speedup of about 3. Another interesting benchmark is *Large* which generates a small and balanced number of relatively deep branches—an ideal situation for the splitting approach. Muse obtains marginal speed-up, while stack-splitting produces considerably better speedups. Better parallel behavior is obtained for almost all benchmarks, except *FQueens*. This benchmark generates a single chioce point with a very large number of alternatives, and each alternative is small and leads quickly to success or failure. In this case stack-splitting pays the price of a slightly more expensive sharing phase. Nevertheless this is clearly not the kind of situations in which a distributed execution is desired.
In the current prototype we have also explored the use of different strategies for splitting the work available in one branch of the computation tree. The graphs in Fig. 7 compares *horizontal* stack-splitting with *vertical* splitting for some of the benchmarks.

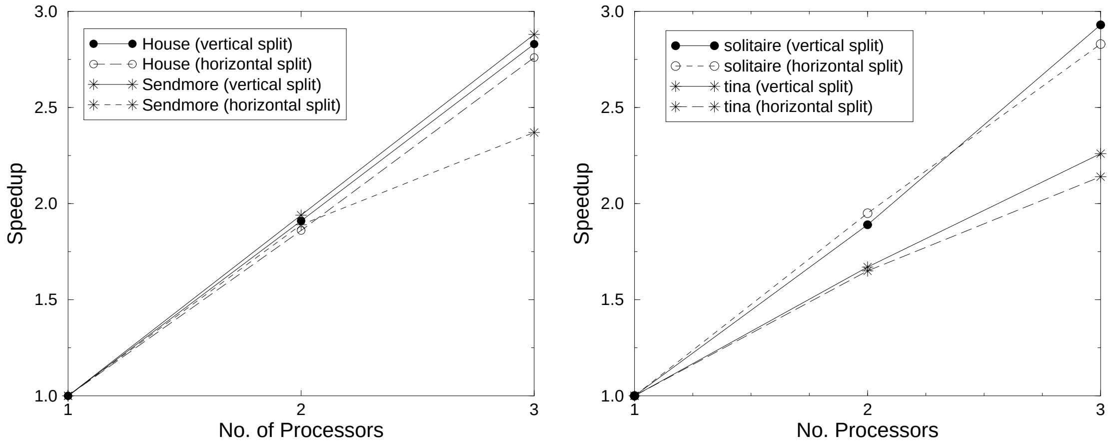

Figure 7: Basic Stack-splitting vs. Vertical Stack-splitting

We are currently carrying out a low-level performance study of our SMM implementation of stack-splitting, using the SimICS Sparc simulator [19] and other performance analysis tools (e.g., Solaris Analyzer). The goal of this study is to determine caching behavior, locality of access, etc., in order to assess the performance one can expect from a distributed implementation of stack-splitting. The preliminary results from this study have already produced encouraging results. For SimICS, we have executed simulations with data cache size of 8MB, and cache lines of both 4 bytes (i.e., single word) and 4K (corresponding to Solaris' memory page size). Fig. 8 compares the largest difference in cache miss rate observed while executing different benchmarks. This shows that the worst case cache miss rate using stack-splitting is considerably lower than the one observed for Muse. Similar results have been observed by taking the best case miss rate.

Figure 10 compares the average number of cache-misses for different benchmarks using increasing number

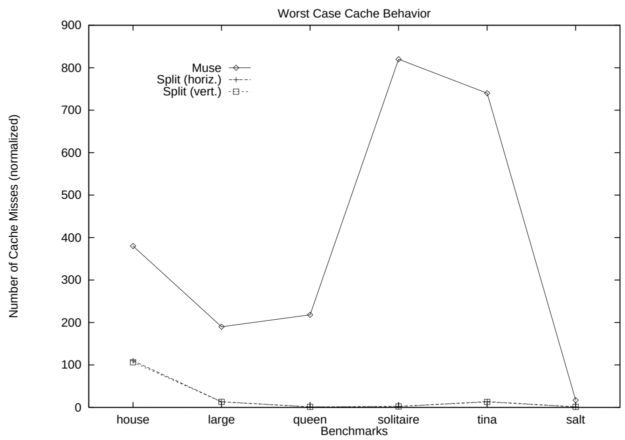

Figure 8: Worst Case Miss Rate

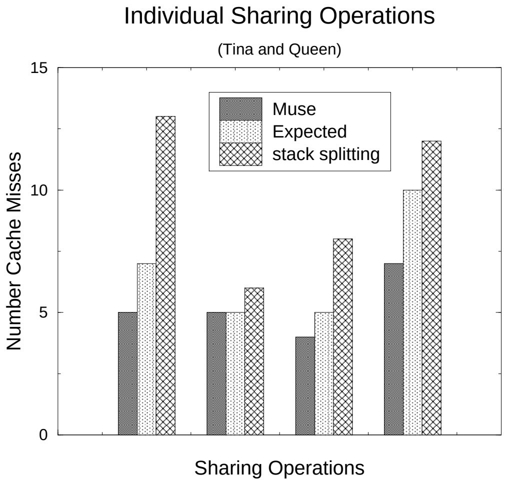

Figure 9: Cache Misses during single sharing

of processors. The number of misses generated by vertical stack-splitting is usually lower than the corresponding numbers obtained from Muse: for House, Large, Salt, and Solitaire the misses are considerably lower, for Tina thye are comparable to Muse, while for FQueens they are worse. Note that horizontal splitting does not perform as well, and this is primarily because most parallel choice-points have only two alternatives (see Section 3.6). In fact, a major factor that leads to increased cache misses in our prototype implementation of stack-splitting is the less-than-optimal implementation of incremental copying. This is borne out by data shown in Figure 9, in which the cache-misses observed in some *individual sharing operations* are plotted: in a sharing operation stack-splitting suffers considerably more cache-misses than Muse—the bar labeled "expected" shows the number of misses we were approximately expecting in stack-splitting based on the status of the caches. Our ongoing quantitative performance studies show that if the number of cache-misses during incremental copying in stack-splitting were identical to that in Muse, then the overall cache-misses in stack-splitting would be considerably lower than that of Muse.

The goal of our low-level performance study is to assess how well a distributed implementation of stack-splitting will perform. Our performance study indeed convinces us that a distributed implementation will perform quite well (provided incremental copying is fully incorporated in the implementation). At this point it should also be mentioned that the cache-miss rate is not an accurate estimate of the number of messages exchanged. The number of cache-misses is an overestimate of the network traffic, since a cache miss is caused either by the "local" behavior of the execution (such as a missing data-item because of an insufficiently large cache) or by the "global" behavior of the execution (data being marked as invalid in the cache of one processor by another). Cache misses due to the latter reason are more reflective of the message traffic in a DMM, as cache misses due to the former can be eliminated by having a very large memory in each node of the DMM. This estimate of the purely global behavior can be obtained by measuring the number of cache invalidates performed during the execution. Fig. 11 compares the average number of net invalidates observed in the different benchmarks. Here, once again, vertical splitting seems to perform quite well. These figures should also improve once sophisticated incremental copying techniques found in Muse are incorporated in our prototype.

An implementation of the stack-splitting method is in progress on a distributed network of shared-memory multiprocessors (on a Beowulf Myrinet-based system with Pentium-II nodes). From the analysis and discussion presented above, it is apparent that stack-splitting should perform well on distributed memory machines, primarily because of better locality and because of need for reduced communication.

6 Conclusion and Related Work

In this paper, we presented a technique called stack-splitting for implementing OP and discussed its advantages and disadvantages. Stack-splitting is a slight modification of stack-copying. Its main advantage,

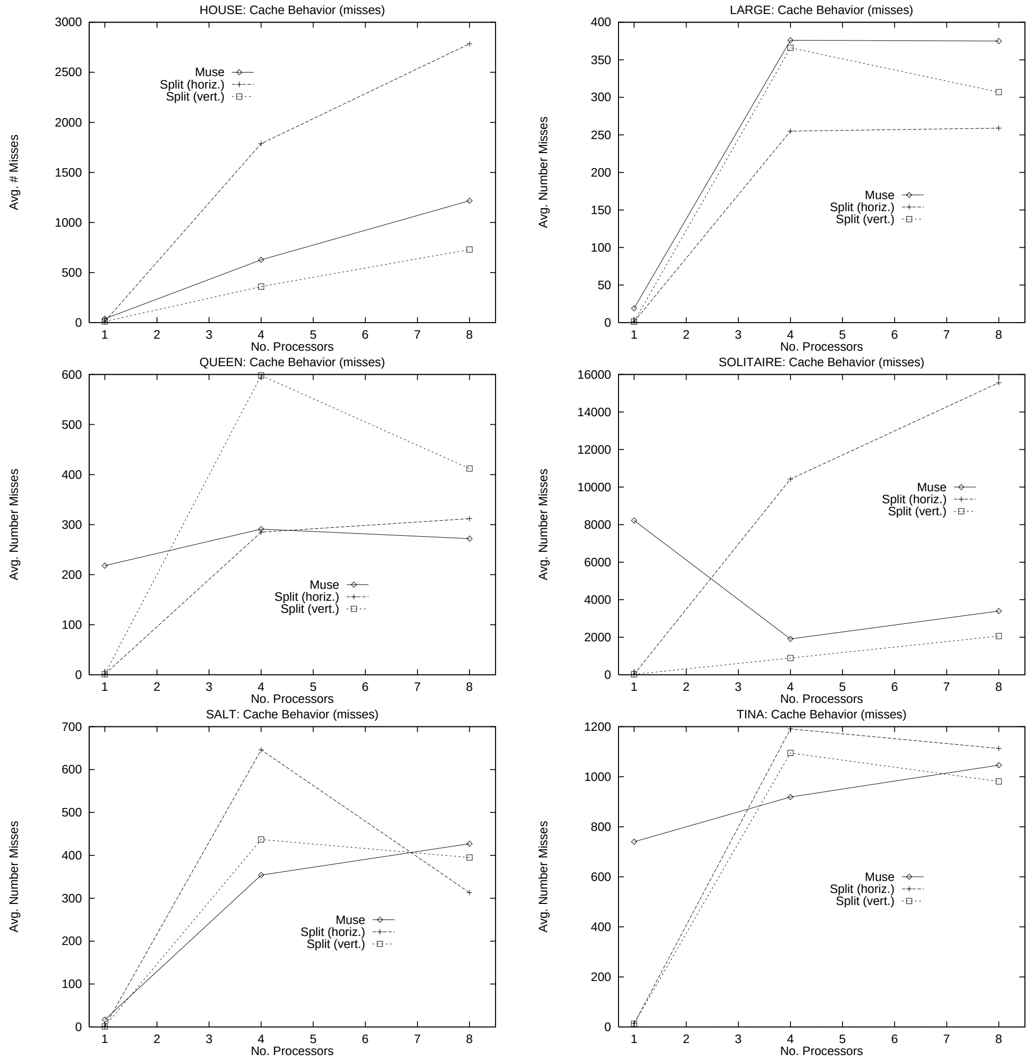

Figure 10: Average Number of Misses

compared to other well-known techniques for implementing OP, is that it allows large grain-sized work to be picked up by idle processors and be executed efficiently without incurring excessive communication overhead. The technique bears some similarity to the Delphi model [9] (it should be noted, though that the Delphi model was not the inspiration for our stack-splitting technique) where computation leading up to a goal with multiple alternatives is replicated in multiple processors, and each processor chooses a different alternative when that goal is reached, resulting in or-parallel execution. Instead of recomputing we use stack-copying, which, we believe, is more efficient. We also showed how the technique of stack-splitting can be used for implementing independent AP.

Distributed implementations of Prolog have been proposed by several researchers [11, 2, 8]. However, none of these systems are very effective in producing speedups over a wide range of benchmarks. Foong's system [11] and Castro et al's system [8] are based directly on Muse-style stack-copying and generate a lot of communication overhead due to the shared choice-points (no real implementation exist for the two of them). Araujo et al's system uses recomputation [9] rather than stack-copying. Using recomputation for maintaining multiple environments for OP is inherently inferior to stack-copying because, essentially, the stack frames that are copied in the stack-copying technique capture the effect of a computation. In the recomputation technique these stack-frames are reproduced by re-running this computation. A computation may run for hours, yet produce only a single stack frame (e.g., a huge tail-recursive computation). Clearly, in such a case (as well as in general) recomputation is inferior to stack-copying. Thus, an approach based

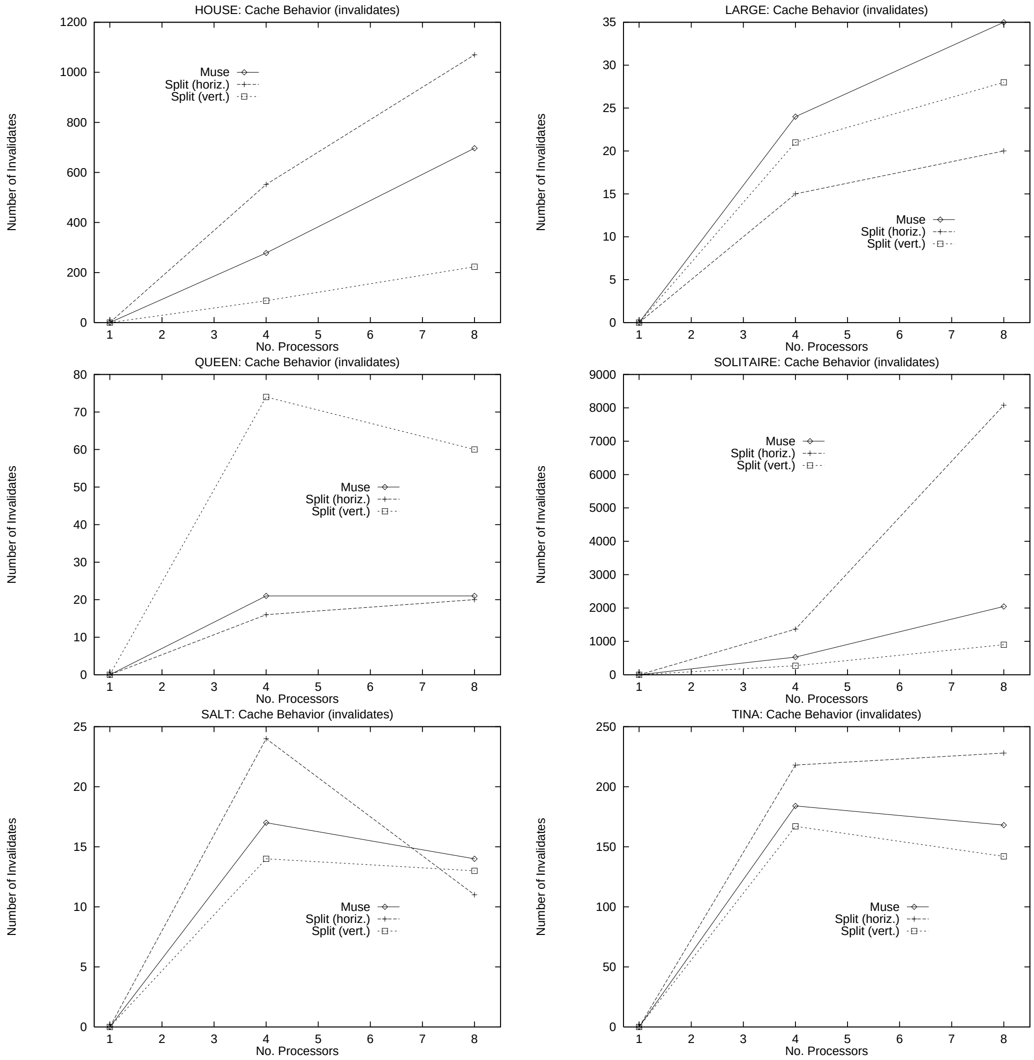

Figure 11: Average Number of Net Invalidates

on recomputation affects the efficiency of a parallel system much more than a stack-copying based system. For this reason, we believe that Araujo et al's approach is not very promising. Distributed implementations of Prolog have been developed on Transputer systems (The Opera System [7] and the system of Benjumea and Troya [6]). Of these, Benjumea and Troya's system has produced quite good results. However, both the OPERA system and the Benjumea and Troya's system have been developed on now-obsolete Transputer hardware, and, additionally, both rely on a stack-copying mechanism which will produce poor performance in programs where the task-granularity is small. We believe that our distributed implementation of Prolog based on stack-splitting will be superior to these aforementioned distributed implementations. A distributed parallel implementation of Prolog based on stack-splitting, with ALS Prolog system (www.als.com) as the underlying engine, is currently in progress. Finally, the idea of stack-splitting bears some similarities with some of the loop transformation techniques which are commonly adopted for parallelization of imperative programming languages, such as loop fission, loop tiling, and index set splitting [30].

Acknowledgements

Thanks to C. Geyer and L. Castro of UFRGS, and to V. Santos Costa for discussions on implementation of distributed LP systems. The authors thank P. Magnusson and Virtutech for providing the SimICS simulator.

References

[1] K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1991. 19(6):445–475.

[2] L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *J. Logic Programming*, 1998.

[3] H. Babu. Porting muse on ipsc860. Master's thesis, New Mexico State University, 1996.

[4] D.A. Bader and J. JaJa. SIMPLE: a Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. Technical report, University of Maryland, 1997.

[5] T. Beaumont and D.H.D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *10th Int'l Conference on Logic Programming*, pages 135–149. MIT Press, 1993.

[6] V. Benjumea and J.M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *Int. Symp. Progr. Lang. Implementations and Logic Progr.*, Springer Verlag, 1993.

[7] J. Briat et al. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, J. Wiley & Sons, 1992.

[8] L.F. Castro, C. Geyer, M. Correia, V. Santos Costa, and F. Silva. DAOS: Distributed And-Or in Scalable Systems. Technical report, Federal University of Rio Grande del Sul, 1998.

[9] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[10] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *Int. Symp. on Logic Programming*, IEEE Computer Society, 1987.

[11] W-K. Foong. *Combining and- and or-parallelism in Logic Programs: a distributed approach.* PhD thesis, University of Melbourne, 1995.

[12] G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Press, 1994.

[13] G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.

[14] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4), 1993.

[15] G. Gupta and E. Pontelli. Last Alternative Optimization for Or-parallel Logic Programming Systems. In *Eight International Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 1996.

[16] G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *Int. Parallel Processing Symposium.* IEEE Computer Society, 1997.

[17] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

[18] E. Lusk et. al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3), 1990.

[19] P.S. Magnusson et al. SimICS/sun4m: a Virtual Workstation. In *Usenix Annual Techn. Conf.*, 1998.

[20] J. Montelius and K. Ali. A Parallel Implementation of AKL. *New Generation Computing*, 14(1), 1996.

[21] E. Pontelli. *High-Performance Parallel Logic Programming.* PhD thesis, New Mexico State Univ., 1997.

[22] E. Pontelli and G. Gupta. *A Simulation Study of Distributed Execution of (Constraint) Logic Programs.* Internal Paper, 1999.

[23] E. Pontelli and G. Gupta. On the Duality Between And-parallelism and Or-parallelism. In *Proc. of Euro-Par'95.* Springer Verlag, 1995.

[24] E. Pontelli and G. Gupta. Parallel Symbolic Computation with ACE. *Annals of AI and Math.*, 1997.

[25] E. Pontelli and G. Gupta. Efficient backtracking in and-parallel implementations of non-deterministic languages. In *International Conference on Parallel Processing.* IEEE Computer Society, 1998.

[26] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Or-Parallelism. In *New Generation Computing*, 17(3), 1999.

[27] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. ACM PPoPP*, ACM Press, 1991.

[28] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-parallel Scheme. In *Proc. Joint Int'l Conf. and Symp. on Logic Prog.*, pages 717–731. MIT Press, 1992.

[29] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Symp. on Logic Prog.*, pages 92–102, August 1987.

[30] M. Wolfe. *High Performance Compiler for Parallel Computing.* Addison Wesley, 1996.