Architectural Support for the Management of Tightly-Coupled Fine-Grain Goals in Flat Concurrent Prolog

Leon Alkalal
Jet Propulsion Lab., MS 198-231
Caltech, Pasadena, CA 91109

Tomás Lang and Miloš Ercegovac
Computer Science Department
University of California Los Angeles, CA 90024

ABSTRACT
We propose architectural support for goal management as part of a special-purpose processor architecture for the efficient execution of Flat Concurrent Prolog. Goal management operations: halt, spawn, suspend and commit are decoupled from goal reduction, and overlapped in the Goal Management Unit. Their efficient execution is enabled using a Goal Cache. We evaluate the performance of the goal management support using an analytic performance model and program parameters characteristic of the System's Development Workload. Most goal management operations are completely overlapped, resulting in a speedup of 2. Higher speedups are obtained for workloads that exhibit greater goal management complexity.

1 Introduction
Flat Concurrent Prolog (FCP) is a high-level, parallel programming language [10] whose syntax and semantics is characteristic of a group of flat committed-choice logic programming languages [5], [7], [16], [11]. The main unit of concurrency in FCP is a goal. An FCP program typically creates numerous concurrent goals that have to reduce for the program to successfully terminate. A goal can create new goals, communicate with other goals using shared logical variables as asynchronous communication channels, and terminate execution. The scheduling of goals is non-deterministic, and their synchronization is performed by sending and receiving messages that result in goal-suspension and data-driven goal-activation.

Preliminary performance evaluation of a distributed implementation of FCP written in Occam and simulated on a system of Transputers [4], as well as performance analysis of FCP execution on a hypercube multiprocessor [13], indicate that the overhead of goal suspensions and the locking of distributed data structures result in performance degradation relative to execution on a single processor [14]. It is suggested that a more efficient single processor implementation is necessary.

To propose an efficient single processor implementation of FCP, a detailed analysis of FCP program execution was performed [2]. The analysis determined algorithmic properties of FCP program execution at the sequential abstract machine level, under a workload that is characteristic for systems development and prototyping. The System's Development Workload consists of seven large benchmark programs which include the Logix Operating System, FCP Compiler, FCP Debugger, FCP Program Analyzer and three Simulators.

The analysis reveals the following characteristics of FCP program execution that are relevant to this paper. A goal performs several iterations prior to termination, suspends frequently waiting for data to be communicated by other concurrent goals, and activates goals when the data arrives. A goal may wait for several messages, and several goals may be activated upon the arrival of a single message. An average of 1.3 goal management operations are performed per goal reduction. We characterized a goal as a fine-grain computation tightly coupled with other goals in the program. By fine-granularity, we imply that the average goal reduction granularity ranges from 20 to several hundred single-cycle instructions.

In a general-purpose implementation, the relative execution time of software-implemented goal management increases as goal reduction is more efficient. Moreover, an environment that supports goal reduction using special-purpose instructions may further increase the relative execution time of goal management. For the System's Development Workload and a single-cycle instruction execution environment, almost 50% of the execution time is spent performing goal management operations [2]. These results strongly motivate the use of special-purpose architectural support for goal management.

We propose architectural support for goal management as part of a special-purpose processor architecture for the efficient execution of FCP [1]. Goal management
operations are decoupled from goal reduction operations and overlapped in the Goal Management Unit. The efficient execution of goal management is performed using a Goal Cache that stores recently spawned goals.

In this paper we evaluate the performance of the Goal Management Unit using an analytic performance model. The program parameters of the model are measured using an instrumented version of the Logix Operating System [12], and executing the System's Development Workload. Using the Goal Management Unit, most goal management operations are completely overlapped with goal reduction, and their execution time is reduced from 50% to less than 3%, resulting in a speedup of 2. We also generalize our results for workloads that have different goal management complexities.

In the following section we describe the behavior of goals in FCP. We then describe the special-purpose FCP processor followed by the Goal Management Unit and Goal Cache. Finally we evaluate the overlapped execution of goal reduction and goal management, and generalize our results.

2 FCP Goals

An FCP program is a set of conditional sentences called guarded clauses [10] that have the following form: \( H \leftarrow G \mid B \), where \( H \) denotes the head of the clause, \( '\leftarrow' \) the implication operator, the guard \( G \) represents a set of conditional goals, \( '\mid' \) is the commit operator and \( B \) the set of body goals. The declarative reading of a guarded clause implies that goal \( H \) can be reduced to a new set of goals \( B \), given that all of the condition goals \( G \) are previously satisfied. Consider the actions of goal \( a \) that consists of the following clauses:

\[
\begin{align*}
a &\leftarrow C_1, C_2 \mid b, c, d. \\
a &\leftarrow C_3, C_4 \mid a. \\
a &\leftarrow C_5 \mid .
\end{align*}
\]

The clauses denote three alternative actions that can take place when goal \( a \) executes. Either \( a \) reduces to goals \( b, c \) and \( d \) if conditions \( C_1 \) and \( C_2 \) are satisfied; or to goal \( a \) if conditions \( C_3 \) and \( C_4 \) are satisfied; or if condition \( C_5 \) is satisfied, goal \( a \) reduces without creating new goals, that is, terminates execution. Note that the second clause represents the iteration of goal \( a \).

The execution mechanism that distinguishes FCP from non-committed-choice logic programming languages is that only one clause may be used to reduce goal \( a \). For example, if all conditions \( C_1, ..., C_5 \) in the above example are satisfied, any one of the clauses may be used to reduce goal \( a \). However, once execution commits to one clause, the possibility of using alternative clauses to reduce the same goal is not considered. That is, only the forward continuation of applied clauses is permitted in FCP. Other logic programming languages like Prolog support both forward and backward program continuations.

In FCP, user annotation of shared logical variables is used to discriminate between the sender and receiver of a message. Consider two goals \( b(X?) \) and \( c(X) \) that share the variable \( X \). The read-only annotation of \( X \), denoted as \( X? \), implies that goal \( b \) can not send a message via channel \( X \), but only receive. However, goal \( c \) that shares the writable version of the channel can send a message. If, due to non-deterministic goal scheduling, goal \( b \) attempts to receive a message before goal \( c \) sends it, goal \( b \) will suspend execution until the message is received. A similar synchronization mechanism to support the parallel execution of tasks is described in [6].

In Figure 1, we show a system of concurrent goals: goal \( a \) reduced to goals \( b, c \) and \( d \); \( e \) sent three messages to goal \( f \); goal \( p \) is suspended on variable \( Z \); and goal \( q \) is suspended on both variables \( Y \) and \( Z \). Goal suspension is implemented using a linked list of suspension records (sr) for each suspending variable. Also, activation records (ar) are allocated for each suspended goal to prevent the same goal from being activated more than once [8]. This enables a goal to suspend on several variables, and several goals to suspend on a single variable.

Goal suspension occurs as a result of a goal reduction attempt that does not fail or succeed. Therefore, goal suspension must occur only if no clause can be used for immediate goal reduction, but some clause may be applicable in the future, when more data becomes available. If there are several clauses that are applicable to the current goal, suspension occurs only after attempting all of them, and finding that none of them succeeded.
3 FCP Processor Organization

We propose an FCP processor architecture that consists of multiple functional units for the execution of FCP programs [1]. It is characterized by a high-bandwidth memory which enables the concurrent manipulation of three types of objects: goals, tagged-data, and instructions. The hierarchical structure of the processor architecture contains tightly coupled execution units, specialized cache units and dedicated memory modules, as shown in Figure 2.

The Reduction Unit, RU, is the main instruction-set unit in the FCP processor. It executes a RISC instruction set that supports pointer dereferencing, multi-way branching and the allocation of data structures on the heap. RU is tightly coupled with the Tag Unit, TU, which concurrently performs tag decoding, setting and extraction. The goal management system supplies RU with reducible goals, and efficiently executes all goal management operations. It consists of the Goal Management Unit (GMU), Goal Cache (GC) and Goal Memory.

The Instruction Unit, IU, provides the execution units with executable instructions. A single instruction contains separate opcode fields for each functional unit. The functionality of the execution units is partitioned so that RU manipulates program data structures, TU manipulates data tags, GMU manipulates goal structures, and IU manipulates instructions. The processor has dedicated cache units that enable faster access to objects requested by the execution units. Objects are requested from the memory modules only on a cache miss.

Memory is also divided into dedicated sections, accessed and managed only by the corresponding execution unit. The Data Memory is used for storing all program data structures such as lists, variables, tuples and constants; Tag Memory is used to store all the data tags; Instruction Memory stores processor instructions and Goal Memory stores all control structures used for goal creation, suspension, activation and termination.

RU-GMU Overlapped Execution

In an FCP implementation that is based on the sequential abstract machine executing on a general-purpose physical machine, FCP programs are translated to a sequence of goal reduction operations, interleaved with high-level goal management functions. Both goal reduction and goal management operations are thus executed sequentially. In the processor architecture that we propose, goal management operations are decoupled from goal reduction instructions using a Suspension Table (ST) and a Wakeup Queue (WQ) as shown in Figure 3. During a goal reduction, RU stores in ST addresses of those (read-only) variables that the current goal may suspend on. At goal suspension, RU switches to an alternate ST, thus allowing GMU to access the old ST and perform overlapped goal suspension. A similar policy applies to the use of WQ: during goal reduction RU stores in WQ pointers to goals that may be activated if the current clause commits, so that, if it does, goal reduction switches to an alternate WQ while GMU activates goals based on the old WQ [3].

GMU executes four high-level goal management instructions: halt, spawn, suspend and commit. Meanwhile, RU continues to execute subsequent instructions, fetched by IU. If, prior to the termination of the current GMU operation, another goal management instruction is fetched, RU blocks until GMU completes its operation.

4 Goal Management System

The purpose of the goal management system is to reduce the contribution of the goal management execution time
to the program execution time. This is achieved by using the Goal Management Unit to overlap goal management and goal reduction, and using a Goal Cache to efficiently implement goal management.

### 4.1 Goal Management Unit

As shown in Figure 4, GMU consists of a Goal Management Controller (GMC), Goal Memory Port (GMP) and Goal Memory Management Registers (GMR). The execution of goal management instructions consists of managing goals stored in the Goal Cache (GC).

GMC uses a Busy Flag control signal to synchronize RU and GMU execution. When GMU receives an instruction from IU, the GMU Busy Flag is set. Subsequent instructions (fetched by IU) that contain goal management instructions are blocked until the flag is reset. When GMU completes the execution of the current instruction, it resets the flag so that any pending instruction is then resumed. The separate memory port, GMP, enables the transfer of goal control structures between GMU and Goal Memory.

GMU contains five special-purpose registers used for the dynamic management of Goal Memory: the Heap Pointer (HP) is used to allocate structures on top of the heap; the Goal Free List (GFL) and the Suspension Free List (SFL) registers store pointers used for reclaiming discarded control structures; and GQF and GQB registers store pointers to the front and the back of the goal queue in memory.

### 4.2 Goal Cache

A goal is represented as a record that consists of a program counter and goal argument pointers. For the System's Development Workload, a goal has on the average 4.8 arguments. If we include the program counter and additional control information, a fixed goal size of 10 words would be sufficient for approximately 95% of all goal record structures [2]. The FCP compiler detects if a goal has more than the maximum number of arguments, and compacts the remaining arguments into a single complex data structure that is dynamically expanded [8]. Having the same size for all goal record structures makes dynamic memory management simpler.

The Goal Cache, shown in Figure 5, consists of N goal windows for storing goals, Goal Status Bits (GSB) that denote the state of the goals, and goal Window Pointers (WP) that point to specific windows. Each window is implemented using (10) words.

The status of a goal window can be one of the following four states. An active window contains the currently executing goal; a spawn window is used for spawning a new goal; a ready window contains a goal that is ready for execution; and a free window is empty. During program execution, there is only one active and one spawn window in the Goal Cache, and they are both addressable by RU. The remaining windows can be ready or free, and are addressable by GMU.

The four registers CAP, CSP, NAP and NSP, contain pointers to the Current Active, Current Spawn, Next Active and Next Spawn windows respectively. CAP is used by RU to access the currently active goal, and CSP to spawn a new goal. If GC contains several ready goals, goal-switching consists of changing CAP to an alternative ready window. Similarly, spawning a new goal consists of changing its status from spawn to ready, while terminating an active goal consists of changing its status from active to free.

The next goal pointer values NAP and NSP are always set by GMU, while RU reads them. The efficient interpretation of goal management operations, as seen by RU, is performed by managing the four goal pointers. To interpret goal termination or suspension, RU moves NAP...
to CAP while CSP remains the same. By prefetching an instruction from the next active goal stored in the window pointed to by NAP (and if there is no wait delay due to RU-GMU synchronization), the goal management instruction is completely overlapped, without contributing to the total program execution time. While RU starts to execute the newly scheduled goal, GMU performs the goal management operation and then determines the new value for NAP. For the spawn instruction, NSP is moved to CSP whereas CAP remains the same. The commit instruction, however, does not affect the goal pointer values.

Two exceptional conditions are detected in the Goal Cache. First, GC-overflow occurs when the goal cache becomes full and there is no free window for spawning. Upon overflow, a goal is moved to the goal queue in Goal Memory. To implement efficient spawning of new goals, overflow is detected before the Goal Cache becomes completely full. That is, there is always one empty goal window available for fast spawning.

The second exceptional condition is GC-underflow, which occurs after a sequence of goal terminations depletes the Goal Cache of available goals for scheduling. To implement efficient halting, there is always at least one prefetched goal available in the Goal Cache. This enables a new goal to be scheduled whenever the current goal terminates execution. Otherwise, RU may need to wait until a new goal is fetched from the Goal Memory.

Those goal management instructions that are implemented by manipulating only the Goal Status Bits in the Goal Cache, are considered cache hits whereas, all goal management instructions that require access to the Goal Memory are considered cache misses.

5 Performance Evaluation

We present a simplified version of the analytic performance model of RU-GMU execution [1]. First we define the performance measures, followed by a description of the system organization and workload, and finally the performance analysis.

5.1 Performance Measures

We define the following two performance measures:

1. The average RU-GMU wait time per executed instruction, $W$, is a measure of the time that RU waits for GMU. The wait time directly affects program execution time and the objective is to reduce it.

2. The GMU (RU) utilization, $U_{gmu}(U_{ru})$, determines the fraction of time spent performing goal management (reduction) relative to the program execution time. Together, the utilization factors are a measure of the workload balance.

5.2 System Organization and Workload

In Figure 6 we show the RU-GMU system organization used to define the performance model. It shows the functional units RU and GMU, the Goal Cache and the Goal Memory. RU executes a RISC instruction set whereas GMU executes goal management instructions. GMU instructions that execute only in the Goal Cache are modeled as cache hits, whereas those instructions that require access to Goal Memory are goal cache misses.

The program parameters used in the performance model are characteristic of the System's Development Workload. These parameters are obtained by instrumenting the Logix Operating System at the abstract machine level, and running existing, large applications.

5.3 Performance Model

In this simplified performance model, we make the following assumptions:

1. The average RU instruction execution rate is one per cycle. We do not take into account that the average execution time can be greater than one due to misses in the Data Cache, branches in the processor pipeline, etc. This assumption is reasonable since there are standard techniques that can be used to bring the execution rate close to one instruction per processor cycle (for example delayed branch).

2. The absolute execution time of a goal management instruction that results in a cache hit is one cycle, since it only requires changing the goal status bits.
3. All goal management instructions that result in a Goal Cache hit are completely overlapped with goal reduction execution, and do not result in wait time. Therefore, their effective execution time is zero.

We define the following performance model program parameters: the total number of GMU and RU instructions is $N_{gmu}$ and $N_{ru}$ respectively; the frequency of GMU instructions is $F_{gmu}$; the average goal size is $S$; the average number of variables per goal suspension is $N_{sar}$; and the average number of activations per goal commit is $N_{act}$. Implementation dependent parameters include the Goal Memory bandwidth $B$, and the Goal Memory word size $L$.

**Average RU-GMU Wait Time, $\overline{W}$**

The average RU-GMU wait time, $\overline{W}$, is equal to the ratio of the total wait time $W$ and the number of executed instructions $N_{ru}$. That is,

$$\overline{W} = \frac{W}{N_{ru}} = F_{gmu} \times \overline{W}_{gmu}$$

where $F_{gmu}$ is the frequency of GMU instructions, and $\overline{W}_{gmu}$ is the average wait time per executed GMU instruction. Since each fetched instruction contains opcode fields for both RU and GMU, the total number of executed instructions is equal to $N_{ru}$. The frequency of GMU instructions then expressed as $F_{gmu} = \frac{N_{gmu}}{N_{ru}}$.

To determine $W$, we consider the execution of two consecutive GMU instructions. Let $d_i$ denote the elapsed time between two consecutive GMU instructions $gmui$ and $gmui+1$, and $t_i^{gmu}$ the time it takes to execute GMU instruction $i$ that results in a Goal Cache miss. If the duration of GMU instruction $i$ is less than the distance $d_i$, the wait time, $w_i$, is equal to zero. However, if the duration is greater than the distance, a non-zero wait time is incurred. Thus, we express the wait time for the $i^{th}$ GMU instruction as

$$w_i = \begin{cases} (t_i^{gmu} - d_i) & \text{if } d_i < t_i^{gmu} \\ 0 & \text{otherwise} \end{cases}$$

The total RU-GMU wait time is then represented as the sum over all executed GMU instructions, that is,

$$W = \sum_{i=1}^{N_{gmu}} (w_i)$$

Note that $t_i^{gmu}$ is the absolute execution time of the $i^{th}$ GMU instruction, and the wait time $w_i$ is its effective execution. Therefore, the average effective execution time of GMU instructions is also the average wait time per goal management instructions. The absolute execution time of the goal management instructions that result in a goal cache miss is shown in Table 1, and is determined by inspecting the goal management algorithms.

<table>
<thead>
<tr>
<th>GMU Instruction</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{halt}^{gmu}$</td>
<td>$[L \times (S + 3)/B] + [L/B + 1]$</td>
</tr>
<tr>
<td>$t_{sp}^{gmu}$</td>
<td>$[L \times (S + 2)/B] + 2((2L/B)) + [N_{act}^{gmu}(3 \times [L/B] + [2L/B] + 2)]$</td>
</tr>
<tr>
<td>$t_{com}^{gmu}$</td>
<td>$[N_{act}^{gmu}(3 \times [L/B] + [L/B])] + 4$</td>
</tr>
</tbody>
</table>

**Table 1: GMU Instruction Execution Times**

The halt and spawn instructions move the goal to memory and perform garbage collection or memory allocation. The execution time of suspend is linearly proportional to the number of variables, $N_{sar}$, the goal suspends on, and the execution time of commit is linearly proportional to the number of goals activated at commit time, $N_{act}$. The suspend instruction always results in the transfer of the suspended goal to the memory, thus effectively behaving as a goal cache miss.

**RU and GMU Utilizations, $U_{ru}$, $U_{gmu}$**

We define the RU utilization $U_{ru}$, as the ratio of the RU execution time, $T_{ru}$, and the total program execution time, $T$. That is,

$$U_{ru} = \frac{T_{ru}}{T} = \frac{1}{1 + \overline{W}}$$

GMU utilization $U_{gmu}$ is similarly defined as the ratio of the GMU execution time, $T_{gmu}$, and the total program execution time $T$. That is,

$$U_{gmu} = \frac{T_{gmu}}{T} = \frac{F_{gmu} \overline{W}_{gmu}}{(1 + \overline{W})}$$

The average absolute GMU instruction execution time $\overline{t}_{gmu}$ is expressed as the sum of the products of the average execution times for the halt, spawn, suspend and commit instructions, and their corresponding frequencies. That is,

$$\overline{t}_{gmu} = F_{halt} \overline{t}_{halt} + F_{sp} \overline{t}_{sp} + F_{susp} \overline{t}_{susp} + F_{com} \overline{t}_{com}$$

If we denote the goal management instruction execution times for a goal cache hit and miss as $\overline{t}_{halt}^h$ and $\overline{t}_{halt}^m$ respectively, the average execution time of instruction $j \in \{halt, spawn, suspend, commit\}$, $\overline{t}_j$, is:

$$\overline{t}_j = F^h_j \overline{t}_j^h + (1 - F^h_j) \overline{t}_j^m$$
5.4 Performance Model Analysis

Average RU-GMU Wait Time, $\bar{W}$

To analytically determine the average RU-GMU wait time under the System's Development Workload, we first determine the distribution of RU instruction distances, $d$, between two consecutive GMU instructions. In Figure 7 we show these distributions for the goal management instructions: halt, spawn, suspend and commit. Using these distributions and the absolute GMU instruction execution times, we determine the RU-Wait time $W$, shown in Figure 8, for three different cases of goal memory bandwidth and two cases for the goal cache size.

The goal memory bandwidth considered is 2, 4 and 8 bytes/cycle. The two goal cache sizes are a minimal cache that consists of 4 windows: active, spawn ready and free, and a goal cache that is large. A large goal cache enables all halt and spawn instructions to always execute in the goal cache. We do not correlate the actual goal cache size and the captured locality of halt and spawn, but just examine the two extreme cases.

In the first column of Figure 8 we show the average wait time $\bar{W}_1$ when goal management operations are implemented in software, that is, without GMU and GC. $\bar{W}_2$ represents the average wait time when the goal management operations execute sequentially using the same execution times as if a minimal goal cache is used. The third and fourth column represent the average wait times when goal management operations are overlapped using a minimal cache $\bar{W}_3$ and a large cache size $\bar{W}_4$.

If the goal management operations are implemented in software, using RU instructions, the average wait time is $\bar{W}_1 = 1$ cycle. That is, half the time is spent performing goal management. Using the goal cache and the goal management algorithms, but not overlapping goal management operations reduces the average wait time almost three fold, for a memory bandwidth of 2 bytes/cycle. That is, $\bar{W}_1/\bar{W}_2 = 3$. The reduction in the average wait time further increases with the goal memory bandwidth.

Overlapping goal management execution using the Goal Cache further reduces the average wait time. For the goal memory bandwidth of 2 bytes/cycle, $\bar{W}_2/\bar{W}_3 = 2.75$. Further reductions in the average wait time are obtained by increasing the goal cache size, but these changes are not significant. The total reduction of the average wait time, from software implementation to overlapped execution using a goal cache is 8.1.

The Goal Cache size, however, does not affect the average execution time of the suspend instruction since every goal that suspends is moved to goal memory. Its execution time depends only on the goal memory bandwidth. The goal cache size influences the average execution time of halt and spawn. Most halt and spawn instructions are overlapped even with a minimum goal cache configuration of 4 goal windows. The difference between the minimum goal cache and a large goal cache becomes even less significant as the goal memory bandwidth increases, since the execution times of halt and spawn are further reduced. Consequently, the wait time is mainly caused by the goal suspension instruction. For the goal memory bandwidth of 2 bytes/cycle, the reduction in the average wait time obtained by increasing the goal cache size is $\bar{W}_3/\bar{W}_4 = 1.8$.

Increasing the goal memory bandwidth reduces the time it takes to transfer a goal to goal memory, which reduces the effective execution time of halt and spawn. Moreover, goal suspension also requires the allocation of suspension lists. Increasing goal memory bandwidth reduces the goal suspension time as long as it affects the transfer time to goal memory. Further increases in memory bandwidth have no effect on the goal suspension execution time. From the results shown in Figure 8 a minimum goal cache of 4 goal windows, together with a goal memory bandwidth of 4 bytes/cycle results in a
RU and GMU Utilization, $U_{ru}$, $U_{gm}$

In Figure 9a we show RU utilization which is over 88%. As the goal cache size or the goal memory bandwidth increase, $U_{ru}$ increases because the average execution times of the goal management operations are reduced, and thus the RU wait time is reduced. This is not the case for the GMU utilization shown in Figure 9b; as the average execution time of GMU instructions is reduced, so is the GMU utilization. This is because the reduction of goal instruction execution time is more significant than the reduction of the resulting wait time.

For the minimum goal cache size and a goal memory bandwidth of 4 bytes/cycle, the GMU utilization is $U_{gm} = 20\%$. This implies an imbalance of goal management and goal reduction leading to an underutilized GMU. One way to improve GMU utilization is to allow GMU to perform additional work instead of being idle. These operations should be of lower priority, so that the requests for goal management are not delayed.

6 Goal Management Complexity

We showed that, for the System's Development Workload, using GMU and GC can double the performance of FCP program execution relative to a software implementation. We now consider workloads that have different granularities of goal reduction and goal management.

Granularity of Goal Reduction

We define the granularity of goal reduction as the average execution time of goal reduction, $T_r$. For a processor that executes one instruction per cycle, the granularity can be measured as the average number of executed instructions per goal, $N_r$.

Granularity of Goal Management

We define the granularity of goal management as the average execution time of goal management, $T_{gm}$, or $N_{gm}$ if measured as the number of executed single-cycle instructions. The System's Development Workload exhibits a higher goal suspension and activation rate than previously reported in [15]. The granularity of goal management, $T_{gm}$, is a function of several program parameters. For example, it depends on the average goal size, the number of variables a goal suspends on, etc. In Table 2, we also show the goal management granularities, $N_{gm}$.

Complexity of Goal Management

The complexity of goal management, $C$, is the ratio of goal management and goal reduction granularities:

$$C = \frac{T_{gm}}{T_r}$$

When we refer to applications with higher complexity of goal management, it means that they have a higher execution time of goal management relative to goal reduction. In Table 2 we show the complexity of goal management for each program in the workload.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_r$</td>
<td>25</td>
<td>43</td>
<td>20</td>
<td>485</td>
<td>728</td>
<td>416</td>
<td>78</td>
</tr>
<tr>
<td>$N_{gm}$</td>
<td>38</td>
<td>139</td>
<td>98</td>
<td>95</td>
<td>56</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>$C$</td>
<td>1.5</td>
<td>3.2</td>
<td>5</td>
<td>0.2</td>
<td>0.1</td>
<td>0.3</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 2: Goal Reduction and Management Granularities
6.1 Overlapped Goal Management vs Complexity

The maximum possible speedup due to overlapped execution of goal management, $S_{\text{max}}$, is achieved when goal management operations are completely overlapped using GMU, resulting in zero wait time. That is,

$$S_{\text{max}} = 1 + \frac{T_{\text{gm}}}{T_r} = 1 + C$$

(9)

Realistically, a delay of $W$ results from the overlapped execution of GMU. Speedup is then represented as:

$$S^r = \frac{T_r + T_{\text{gm}}}{T_r + W} = \frac{1 + C}{1 + \frac{W}{T_r}}$$

(10)

If we consider granularities greater than 200 as high, and 10-30 instructions as low, we then partition the space of goal management and reduction granularities into the following four regions, as shown in Figure 10.

- **M-domain**: Low goal management and high goal reduction granularities are characteristic of Meta-Interpreters, and other applications that perform symbolic interpretation of programs as data.
- **A-domain**: Low goal management and goal reduction granularities are typical of small applications, often used for comparative benchmarking, such as the list Append program. These applications perform very little goal management and the reduction granularity is approximately 20 RISC operations.
- **S-domain**: High goal management and goal reduction granularity was observed in the System's Development Workload.
- **C-domain**: High goal management and low goal reduction granularity is characteristic of applications that explicitly model the inter-goal communication and synchronization. The set of FCP programming stereotypes used in [13] are of this type.

In Figure 10 we show the average program execution time of the System's Development Workload, $T$, which consists of the granularities $T_r$ and $T_{\text{gm}}$. When goal management instructions are implemented in software, we showed that $T_r \approx T_{\text{gm}}$. Therefore, the complexity of goal management for the S-domain is $C \approx 1$, and the maximum speedup is $S_{\text{max}}^S = 2$. In this case, the goal management operations were considered a bottleneck, which motivated special-purpose support using GMU. We showed that the wait time due to overlapped execution can be made small, less than 4% of $T_r$. Therefore, $S_{\text{max}}^S$ is close to the maximum possible, since the operations are almost completely overlapped. Thus, $S_{\text{max}}^S \approx S_{\text{max}}^S \approx 2$.

If we label the goal reduction and goal management times in the C-domain as $T_r'$ and $T_{\text{gm}}'$ respectively, where $T_r' < T_r$, then $C_C > C_S$. Therefore, the maximum possible speedup due to overlapped goal management using GMU in the C-domain, $S_{\text{max}}^C$, is $S_{\text{max}}^C > S_{\text{max}}^S = 2$.

In Figure 11 we show the maximum speedup due to overlapped and efficient execution of goal management operations for each program in the workload. Programs like the Simulator2 spent more time performing goal management than goal reduction. In addition, we considered simple applications that are used to describe specific distributed algorithms and communication protocols. For example, in the Lord of the Rings algorithm that computes the extreme value of nodes connected in a unidirectional circle using $O(N \log N)$ messages [9], a goal reduction consists of executing only a few instructions that check whether the input channel has a message. If it has, a reply message is sent on the output channel, whereas, if the message has not arrived, the goal reduction suspends. The average goal reduction granularity is 30 RU instructions. To compute the goal management granularity, in Table 3 we show the total number of goal management operations and goal reductions for
The speedup due to overlapped and efficient execution of goal management is greater than 4% and a realistic speedup of 5 was obtained, resulting in $S_{\max} = 6$ and a realistic speedup of 5 due to the wait time.

## 7 Conclusion

We propose special-purpose architectural support as a way to reduce the goal management execution time in FCP. The architectural support consists of a dedicated Goal Management Unit that executes high-level goal management operations concurrently with goal reduction. Moreover, the efficient execution of goal management is enabled using a Goal Cache that stores recently spawned goals. Operations such as goal-switching, spawning and halting are efficiently performed by changing their status in the Goal Cache. More complex operations such as suspension and activation are decoupled from goal reduction by using two Suspension Tables and Wakeup Queues. For the System's Development Workload, which consists of large FCP programs, we show, using an analytic performance model, that the overhead of software-implemented goal management is 50% of the program execution time. This is reduced to 4% using the Goal Management Unit and Goal Cache, resulting in a speedup of almost 2. We generalize our results for workloads that exhibit different goal management complexities. Programs that explicitly model inter-goal communication and synchronization exhibit higher goal management granularity than goal reduction. For these programs, the speedup due to overlapped and efficient execution of goal management is greater than 2; for a simple distributed algorithm a speedup of 5 was obtained.

## References


