An evaluation of Penny: a system for fine-grain implicit parallelism

Johan Montelius and Seif Haridi
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
jm, seif@sics.se

Abstract

The Penny system is an implementation of AKL, a concurrent constraint language with deep guards, on shared-memory multiprocessors. It automatically extracts parallelism in arbitrary AKL programs. No user annotations are required nor there is any compiler support to extract parallelism. We give an overview of the system and present empirical evaluation results from a set of benchmarks with varying characteristics. The evaluation shows that it is possible to build a system that automatically exploits fine-grain parallelism for a wide range of programs.

1 Introduction

The Penny system is an implementation of AKL, a concurrent constraint language with deep guards. The system has been implemented on a high-performance shared-memory multiprocessor and is able to outperform C implementations of algorithms with complex dependencies without any user annotations.

In this paper we describe a performance evaluation of the system. Extensive measurements were done using both smaller benchmarks as well as real-life programs. The evaluation uses detailed instruction-level simulation, including cache-performance, to explain the behavior of the system.

Section 2 of the evaluation shows the performance of the system for different classes of benchmarks. The tests include simple recursive, stream parallel and non-deterministic benchmarks. The next two section show the limitations of the system when the granularity of work decreases. In Section 3 a simple recursive program is used while Section **??** uses a fine-grain concurrent program.

Section 5 shows that the system can outperform a C implementation of algorithms with complex dependencies. Section 6 shows how the system performs when running its own compiler. This is a real-life program that has not be optimized for parallel execution.

We conclude that it is possible build a system that automatically exploits fine-grain parallelism in a satisfactory way for a wide class of programs. In order to achieve this we have used the following design strategies:

- task creation for parallel execution is demand driven;
- local scheduling of woken tasks is lazy;
- global scheduling is performed by the idle workers; and
- cache performance is always considered.

We first summarize the basic features of the language and the structure of the implementation. A complete description of the language can be found in [9, 7].

1.1 Agents Kernel Language

An AKL program consists of a set of procedure definitions. Procedures can either be in compositional definitional form, or in clausal form similar to concurrent logic languages. We describe here only the clausal form which is used in the Penny system.

A procedure definition consists of a set of guarded clauses having the same guard operator % and the same procedure name p/n.

p(X_1, ... X_n) :- G % B.

G is called the guard of the clause, and B is the body. The guard and body of a clause are (possibly empty) sequences of goals. A goal is either a constraint, a procedure call, or an aggregate call, (set-of, bag-of, etc). The guard operator % is one of $\rightarrow$, ?, and |, and its corresponding clause is either conditional, nondeterminate, and committed-choice clause respectively. AKL differs from other existing concurrent constraint languages in that the guards may have arbitrary procedure and aggregate calls. This leads to a deep-guard language, having computations that may result in a hierarchy of constraint stores. Penny implements only constraints on rational trees with the equality constraint-operator. Therefore we restrict our description to rational trees.

A computation starts by an initial sequence of goals and an empty constraint store called the main and-node. Executing a procedure call creates, in general, a choice-node having as immediate children an ordered sequence of and-nodes, one for each guard. The order of the and-nodes corresponds to the order of clauses of the procedure being executed. Variables always belong to one and-node. Initially each variable belongs to the node where it is introduced. Variables belonging to an and-node A are visible to all the and-nodes in the subtree rooted at A.

When a constraint is executed it is added to the and-node. A constraint in an and-node A on a nonlocal variable

is called a conditional constraint. A constraint is only visible in the and-nodes in the subtree rooted at A. An and-node may fail if the union of constraints that exist in the path from the root node to, and including, the and-node is inconsistent. Failed and-nodes are removed from their parent choice-node.

Conditional constraints in an and-node that are logically entailed by the union of constraints that exist in the path from the root node to the current and-node are removed from the and-node. A non-failed and-node that is fully executed, i.e. no calls are left in it, is a *solved* and-node. A solved and-node that has no conditional constraints is an *entailed* and-node.

Execution of a procedure call defined by a conditional procedure proceeds by creating the corresponding choice-node, and its and-nodes, and thereafter committing to the leftmost entailed and-node, i.e. committing to the leftmost entailed guard. Execution of a procedure call defined by committed-choice procedure commits to an arbitrary entailed and-node, i.e. to arbitrary entailed guard. In both cases commitment merges the constraint store of the guard with the constraint store of the parent and-node. Also the goals in the body of the corresponding clause replaces the choice-node in the parent and-node; the body is *promoted*. Merging also implies that the variables created during guard execution are promoted to belong to the parent and-node.

An aggregate procedure follows the same rules as a conditional procedure with the exception that the promoted body does not replace the choice-node but is added to the left of the node; the body is *collected*. A collected body is also given access to a variable where the solution can be recorded.

Execution of a procedure call defined by a nondeterminate procedure proceeds by creating a corresponding choice-node, and its and-nodes. The execution commits to an and-node only when there is a single solved and-node left. This is called *determinate promotion*. Notice that there is no requirement that the and-node should be entailed.

The execution rules described so far are the *determinate rules*. When all determinate rules have been applied in a subtree rooted at an and-node A, and no conditional constraints on variable above A exist in the subtree formed by A, then the node A becomes *stable*.

Nondeterminate execution proceeds on a stable and-node A by 1) selecting the leftmost innermost nondeterminate choice-node C, with a parent and-node P, and splitting it into two choice-nodes $C1$ with the first child and-node of C, and $C2$ with the remaining children of C; 2) creating a copy $P1$ (clone) of P with disjoint constraint stores and disjoint variables, $C1$ belonging to the clone $P1$, and $C2$ replacing C in the original and-node P. Since $C1$ holds a single and-node it is now eligible for determinate promotion.

As can be deduced from this brief description, and-parallelism corresponds to parallel executions of goals in various and-nodes, whereas the cloning of and-nodes in a nondeterminate execution step introduces or-parallelism. Parallel execution of different deep guards as well as the execution of aggregate calls, allows multiple search problems to be executed in parallel. This feature is novel in the AKL as well as in the Penny system.

1.2 The Penny abstract machine

The Penny compiler transforms AKL programs to *abstract machine* instructions. The Penny compiler is itself written in AKL, and consists of about three thousand lines of AKL code that can in turn be executed on Penny itself, with good parallel performance. The emulator is relatively small since the current abstract machine only consists of sixty-three instructions. Most of the instructions are register-based WAM style instructions [17].

When a program is executed, a fixed set of *workers* are created. The number of workers determines the level of parallelism, so there is no advantage in creating more workers than the available number of processors. Each worker is dynamically assigned work during an execution. If a worker runs out of work it steals a task from another worker. This approach has successfully been used in other system, for example JAM and MultiLisp [5, 14].

1.3 The execution state

During an execution the workers build and modify a shared *execution state*. The execution state is a tree structure of choice-nodes, and-nodes and *continuations*. The choice-nodes and and-nodes corresponds to those of the AKL computation model and represent goals that are under evaluation. The continuations represent sequences of untried goals.

An and-node represents a guard computation. It holds a mixed sequence of choice-nodes and continuations that represents the goals of the guard, a continuation that represents the body, and a list of bindings that represents the conditional constraints of the and-node. As conditional constraints are quite rare, a binding scheme that keeps these explicitly represented works very well [15].

When a nondeterministic step is performed, an and-node is completely duplicated. No structures are shared between the two copies so there is no need to keep track of different binding environments because of nondeterminism. The binding scheme only needs to keep track of the bindings in different levels in the execution state.

1.4 Scheduling

Each choice-node in the execution state is either ready to be executed or suspended on some AKL variable. In a similar manner, an and-node can be suspended on variables external to the node. Suspended nodes are registered on the variables that cause the suspension. A worker that adds a new binding to a variable also creates a *wake task* for each suspended node that is to be re-scheduled.

When a continuation is executed the first procedure call is selected and removed from the continuation. A continuation task is then created that refers to the updated continuation. If the procedure was the last procedure in the continuation the continuation is removed.

All tasks that were created in an and-node belongs to the and-node and are handled before the worker leaves the and-node. The tasks are kept in two stacks the *wake stack*[1] and the *continuation stack*. Each worker has its own stacks.

When a new task is selected the continuation tasks are given priority over the wake tasks. Tasks are also executed fully i.e. there is no task preemption. This leads to a *lazy* execution strategy where woken nodes are not scheduled for execution as soon as feasible, but rather executed later when a worker runs out of continuation tasks. This lazy strategy has proved to be very stable since it increases the locality of work available to a worker.

[1] The wake tasks are actually divided into tasks that refer to and-nodes and tasks that refer to choice-nodes, these types have separate stacks.

Selecting a new task is in the simple case very cheap. A new continuation task is rescheduled in about the same time as a **proceed** instruction in a Prolog system. A wake task is more expensive since the worker has to *install* itself in the woken and-node.

A worker that runs out of tasks can steal a task from another worker. The busy workers are examined and a task is stolen from the bottom of a task stack. Priority is given to wake tasks over continuation tasks.

1.5 Memory management

The data structures that are used to represent the execution state can be explicitly reclaimed by a worker. This improves cache performance since the same cache-lines can be reused immediately. AKL terms cannot be explicitly reclaimed and are therefore allocated on a *heap*, shared between all workers, that is subject to garbage collection.

A parallel stop-and-copy garbage collector is used. It is important that the garbage collector is parallel since the garbage collection time would otherwise completely dominate the execution time. Not only does the parallel execution decrease the execution time but the amount of garbage may also increase. Therefore, a sequential garbage collector does not only increase the proportional time spent in the collector but also increases the total time spent on garbage collection.

The collector has a hard time keeping up with the overall increased performance but the garbage collection time normally stays well below 10% even for parallel executions.

There are two sources for parallelism in the collector. First of all the execution state is divided among the workers. The workers are then responsible for copying their part. A worker that runs out of work steals work from other workers. If no worker has any nodes available the AKL terms are divided into segments that can be distributed among workers [1]. This kind of parallelism is necessary since some programs use very few nodes in the execution state with few but very large AKL terms.

1.6 The target machine

Our target machine for the implementation is a SPARCCENTER 2000 (SC2000) multiprocessors with 20 processors. The SC2000 is a bus-based shared-memory multiprocessor from Sun Microsystems.

The SC2000's processors, 50MHz SuperSPARCs, have a 4Kbytes direct mapped first level data cache with 32 byte long cache lines [2] The instruction cache is 20Kbytes, 5-way associative with 64 byte cache lines.

The processors connect to an off-chip cache, the second level cache, which on the SC2000 is 2Mbytes, direct-mapped with 64-byte cache lines. These caches are connected to a bus, whereby they can communicate with the main memory and/or other caches with a peak sustainable read/write throughput of 500Mbytes per second [4].

Accessing data that resides in the first level cache is done in one clock cycle. Accesses to the second level takes 5-10 cycles, whereas accessing the main memory takes 20-60 cycles. A high cache hit rate is therefore crucial for good performance. The system has been designed in a way that takes this into account. Data structures have been carefully aligned to avoid false sharing. All data structures that are used to represent the execution state (except AKL terms) use one cache-line each.

The SIMICS instruction level simulator has been used to evaluate the cache performance of the system [13]. The simulator accurately simulates a multiprocessor SPARC architecture and can provide valuable statistics. The read miss figures that are presented in this paper is the ratio between the read operation that miss the second level cache and all read operations.

Each worker is running as a Solaris thread directly mapped to a Solaris LWP [12]. The threads share the same address space and have full access to all data structures in the system, this makes the distribution of tasks very easy. The system currently runs under SunOS 5.4 but should not be hard to port to other operating systems. The emulator is a threaded code emulator implemented in C using the GNU C compiler (version 2.7) where labels can be handled as data.

The execution time does not include the initialization of the system. The clock is started when all memory areas have been allocated, code loaded and all workers prepared to enter the scheduler. The clock was stopped when the threads that implement the workers terminated. All timings are wall time, measured by the Solaris **gettimeofday()** system call.

The initial value for heap and free lists were set so that there would be no overhead for allocating new blocks during runtime. The execution times reported contain, if not explicitly stated, the garbage collection time.

2 Performance

The Penny system is about two to three times slower than emulated SICStus v3. This is when comparing small benchmarks that do not use any of the special features of the SICStus implementation nor of the Penny implementation. This means an execution speed [3] of about 260K LIPS on a Sun SPARCstation 20. Emulated SICStus v3 executes at about 560K LIPS on the same machine.

The slower performance of the Penny system compared to SICStus v3 is mainly due to its unoptimized compiler. If the code is hand optimized to mimic a normal Prolog compiler, the execution speed is about 380K LIPS. This is without changing the existing instruction set. A new compiler for a sequential AKL system shows figures that are as good as emulated SICStus v3. The compiler was not available when the Penny system was constructed.

The parallel performance is harder to evaluate. It is of course very dependent on the application. In this section we examine the parallel performance of the system using a benchmark suite with varying characteristics.

2.1 Simple recursive programs

In the first test we have chosen benchmarks that should show good parallel performance. The benchmarks are simple to analyze and should have a predictable behavior. These benchmarks are a first test of the system. Good performance is a necessary but not sufficient condition for the validation of the system. The benchmarks are often used in the logic programming community.

matrix: A 500×500 matrix of floats multiplied by a vector. The program spawns a sequential task of fixed size in each recursion.

[2] The data cache is normally 16Kbytes, four way set-associative but on the machine we are using only one of the four sets is used.

[3] The old naive-reverse of a thirty element list

Workers	2	4	8	12	16	18
matrix	1.96	3.80	7.39	10.2	12.7	13.8
hanoi	2.05	3.87	7.74	11.1	13.3	14.0
fib	1.99	3.85	7.74	10.7	13.0	13.5
tak	2.02	4.01	7.85	11.1	13.5	14.1

Table 1: Simple benchmarks, speedup

Workers	2	4	8	16
matrix	0.028 %	0.042 %	0.060 %	0.121 %
hanoi	0.007 %	0.007 %	0.013 %	0.024 %
fib	0.006 %	0.007 %	0.024 %	0.111 %
tak	0.006 %	0.009 %	0.030 %	0.210 %

Table 2: Simple, read cache performance

hanoi: The towers of Hanoi using 18 bricks. The program divides into two identical subtasks in each recursion.

fib: The Fibonacci function calculating fib(27). The program divides into two subtasks of different size in each recursion.

tak: The Takeushi function calculating tak(20,10,4). The program creates four subtasks of very different sizes in each recursion.

The Fibonacci and Takeushi benchmarks are numerical benchmarks, so they do not create any data structures other than variables and integers. The matrix benchmark creates lists to represent both the matrix and the vector. The hanoi benchmark creates a list of length 2^{16}.

Figure 1 shows the minimum and maximum execution time of a hundred runs. As clearly seen they all behave well. The maximum execution time is less than 10% higher than the minimum execution time when eight workers are used. The difference is significantly larger (up to 27%) when fourteen workers are used but even then are remarkably stable. When more than eighteen workers are used almost anything can happen but the minimum execution time does still decrease. In the following sections only the execution time up to eighteen workers is reported.

If we look at the speedup figures in Table 1, based on the median execution time, we see that the system performs very well when eight workers are used. The system then performs slightly worse and gives a total speedup of about thirteen to fourteen. If the minimum execution times are compared, the final speedup for twenty workers are for all benchmarks above fifteen.

Table 2 shows the read miss rate of the second level cache. The matrix benchmark has a significantly higher miss rate, this probably pertains to the initial reading of the matrix. The high miss rate for the Fibonacci and Takeushi benchmarks initially very low but then increases when sixteen worker are used. There are, in both benchmarks, dependencies between the arithmetic operations. The dependencies cause suspensions which in turn cause read misses. The hanoi benchmark has the lowest miss rate, this benchmark is completely free of dependencies that could cause suspensions.

The read miss rate are for all benchmarks very low and does not severely limit the obtained speedup. These figures serve as a reference when cache performance is evaluated for the benchmarks in the following sections.

The creation of tasks is different in each program. The binary spawn, in Hanoi, is the most profitable since it very quickly divides the available work into equal size tasks. The scheduler does not guarantee that the initial allocation is perfect but the available work is rapidly balanced as workers run out of tasks. The linear spawning of tasks in the matrix program has a disadvantage that we investigate further in section 3.

Workers	2	4	8	12	16	18
mastermind	1.92	3.81	7.34	10.1	11.9	12.4
kkqueen	1.94	3.81	7.42	10.1	11.2	11.5
turtles	1.82	3.38	5.68	7.00	7.18	7.43
qsort	1.86	3.48	5.67	6.70	6.51	6.37

Table 3: Stream parallel, speedup

2.2 Stream parallel programs

In the stream and-parallel programs there is a flow of information between the goals. Goals can be executed in parallel but there are dependencies between goals that causes goals to suspend. The set of programs that we have chosen for this evaluation can all be executed from left to right without any suspension, i.e. they can all be executed in a Prolog system.

The benchmarks are:

mastermind: The mastermind puzzle by E. Tick, using two guesses and three colors.

kkqueen: The candidates/non-candidates queens program by K. Kumon/E. Tick, using 9 queens.

turtles: The turtles puzzle by E. Tick using layered streams.

qsort: Quick-sort of the first 10.000 four digit numbers extracted from the decimals of π

The benchmarks execute on one processor in between two and four seconds. As shown in Table 3 the system behaves well but not as good as for the simple recursive benchmark in the preceding section. The most noticeable is the limited speedup in the quick-sort and turtles benchmarks.

There is a sequential component in the quick-sort program. Any execution must traverse the initial list once and this can only be done sequentially. The n steps required to partition the initial list sets a lower limit on the execution time regardless of the number of workers. The turtles program has no comparable sequential component that could explain the poor speedup.

The sequential component is however not the limiting factor in these benchmark. If this was the case the speedup would be about twice as good. If we run the benchmarks using SIMICS we get the statistics listed in Table 4.

The read cache performance is clearly worse for the quick-sort and turtles benchmarks. When eight or sixteen workers are used the miss rate is almost a magnitude higher than for the mastermind benchmark. This is significant and severely limits the obtainable speedup.

Notice that the cache miss rate is not improved in a system with explicit parallelism. The "best" annotation in the quick-sort program would allocate the first worker to do the initial partitioning while two other workers would do the recursive sorting. This results in a high miss rate since the

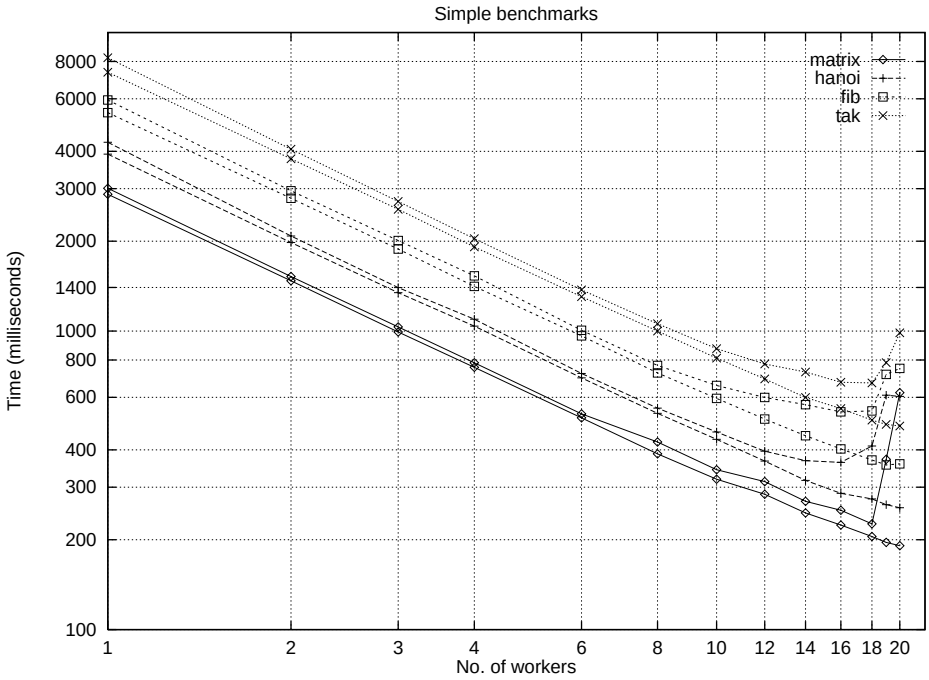

Figure 1: Simple recursive benchmarks

Workers	2	4	8	16
mastermind	0.021 %	0.029 %	0.057 %	0.15 %
kkqueen	0.017 %	0.019 %	0.030 %	0.11 %
turtles	0.11 %	0.19 %	0.37 %	0.65 %
qsort	0.051 %	0.20 %	0.46 %	0.97 %

Table 4: Stream parallel, read cache performance

Workers	2	4	8	12	16	18
queens	1.83	3.40	5.74	6.74	6.49	6.33
f-queens	1.92	3.65	6.91	8.67	8.36	7.88
scanner	1.89	3.46	5.88	6.91	7.00	6.84

Table 5: Or-parallel speedup

recursive sort procedures would have to read the input lists from the cache of the first worker. High cache miss rate is inherent in the stream and-parallel programs and it is not solved by explicit allocations of goals to processors.

2.3 Non-deterministic programs

One of the benefits of AKL is that one can implement both and- and or-parallelism. Or-parallelism can be exploited in all situations where more than one guard of a goal is evaluated. In the Penny system guards are evaluated lazily and are only evaluated in parallel in nondeterministic programs.

Parallel execution in a nondeterministic program is either very simple or very hard. It is very simple if all solutions to a problem are needed. The workers can divide the work in independent parts and do very little communication to synchronize their activities. If only one solution is needed the situation is much harder. If the leftmost solution is wanted and the solution is at the far left in the search tree two worker will not find the solution faster than one worker. The second worker will only do *speculative work*.

In a system that is targeted to or-parallel execution the ability to avoid doing speculative work is crucial [2]. In the Penny system very little has been done to minimize speculative work. The scheduler is in fact unaware of the fact that some task pertain to nondeterministic executions. The speedup for benchmarks where the first solution is wanted can therefore vary from nil to good.

If we are looking for all solutions the speedup is more predictable. The following benchmarks use various kinds of programming techniques to solve puzzles.

queens An implementation of the queens puzzle (10 queens) using a short circuiting technique. The benchmark needs 5904 split operations.

f-queens A simple but very fast Prolog like implementation of the queens puzzle (10 queens). Does not make use of concurrency to minimize the search space. Needs 35538 split operations but the copied state is very small.

scanner Finds a pattern in a grid given information of the number of filled squaes in each line. Uses a advanced short circuiting technique where *ports* are used to communicate between processes [11]. The benchmark needs 2426 split operations.

Table 5 shows the speedup given the median execution time from 40 runs. The benchmark all behave well when using up to twelve workers. They are then stable and drop slightly in performance when eighteen workers are used.

These figures are with garbage collection included, if the garbage collection is excluded the figures become slightly

Workers	2	4	8	12	16	18
queens	1.64	2.20	2.79	3.50	4.07	4.16
f-queens	1.25	1.53	2.55	3.27	3.58	3.70
scanner	1.55	2.93	4.52	4.73	4.04	4.03

Table 6: Multiple nondeterminism, speedup

but not dramatically better. The speedup for the scanner benchmark reaches a maximum of 8.35 if garbage collection is omitted. Since the execution state increases in size as more workers are added (a larger part of the search tree must be explicitly represented) the parallel garbage collector has a hard time keeping up with the increased performance.

For the scanner benchmark this is a sever limitation. The garbage collection time increases from about three-hundred milliseconds to twelve hundred milliseconds. Compared to the total execution time this is an increment from less than a percent to seventeen percent.

When garbage collection is performed the workers have to traverse the whole execution state. The cache performance of this operation is very poor. The scanner benchmarks has a read miss rate of 6.5% when eight workers are used. The miss rate of the first worker, that is responsible for dividing up the execution state, is as high as 11%.

2.4 Multiple nondeterminism

Good speedup could be expected in a program with many smaller nondeterministic calculations. To test this the same three benchmarks were used to solve one hundred puzzles, selecting only the first solution, in parallel. Since each puzzle is encapsulated in a guard they can be solved independently of each other.

As shown in Table 6 the speedup is quite disappointing. The reason for this is that the global scheduler favors wake tasks on the expense of continuation tasks. Since the wake tasks also are the keys to nondeterministic computations the result is that workers collaborate in each computation instead of selecting their own computation.

A solution to the problem would be to give priority to tasks at the main level. This could aid the workers so that they spread out in the execution state. It would of course also mean that the execution state becomes larger which in turn might hamper the garbage collector.

2.5 Deep programs

A Penny program can of course be arbitrary deep. Nondeterministic programs always encapsulates a computation inside a guard but this is not the only reason for working with deep guards. The compiler, for example, uses five levels without doing any search.

When computations are deep, the system needs to be able to distribute tasks at arbitrary levels in the computation. Distribution of a deep task is a complex operation that might reduce the speedup.

To test this feature the knights benchmark was used. This is an implementation of the "touring knight" i.e. place a chess knight in one corner of a board and jump to all squares without touching any square twice. The benchmark searches for all solutions (304) on a five times five board. This benchmark behaves very well and a speedup of twelve is reached using eighteen workers.

Workers	2	4	8	12	16	18
Level 1	9	42	110	169	255	333
Level 2	143	303	1555	3289	6501	6557
Level 3	5	29	168	737	1494	2025
Level 4	-	4	49	133	288	308
Level 5	-	-	15	8	34	35

Table 7: Stolen tasks at different levels

```
multiply([], _, Result):-
        ->      Result = [].
multiply([V|Rest], Vector, Result):-
        ->      Result = [R|Product],
                vmul(Vector, V, 0, R),
                multiply(Rest, Vector, Product).
```

Figure 2: Matrix multiplication

Table 7 shows the number of tasks stolen by workers on the different levels in the execution state. The majority of all tasks are stolen at level two, this is the level where the different solutions are collected, but a substantial amount of tasks are collected at the third level. The collection of tasks at the fourth and fifth level shows that there is parallelism to exploit at all levels.

Scheduling of deep tasks adds to the complexity of the scheduler. A system where only tasks are distributed at the main level would be simpler to implement but would also limit the obtainable speedup. We have chosen to implement a scheduler that handles deep guards since the programmer otherwise would have to be aware of the limitations in the scheduler to get good performance.

3 Granularity

The matrix multiplication benchmark creates a vector and a matrix and then multiplies the vector and the matrix. The definition of the multiplier is shown in Figure 3. A worker that executes the `multiply/3` definition creates a continuation task in each call of `vmul/4`. The continuation task is the key to the parallelization of the process.

To investigate how the granularity of tasks can change the performance, four benchmarks were executed. The benchmarks multiplied a matrix of size $M \times N$ with a vector of length N. The product $M \times N$ is equal for all benchmarks so the number of floating point operations is equal in all benchmarks.

The median execution time of one hundred runs is shown in Figure 3. There are four important observations that can be made:

- the execution time using one worker is almost identical for all benchmarks,
- the execution time using two workers is significantly different,
- the speedup from two to six workers is almost constant, and
- a sharp knee is reached when using 6, 10 and 14 workers.

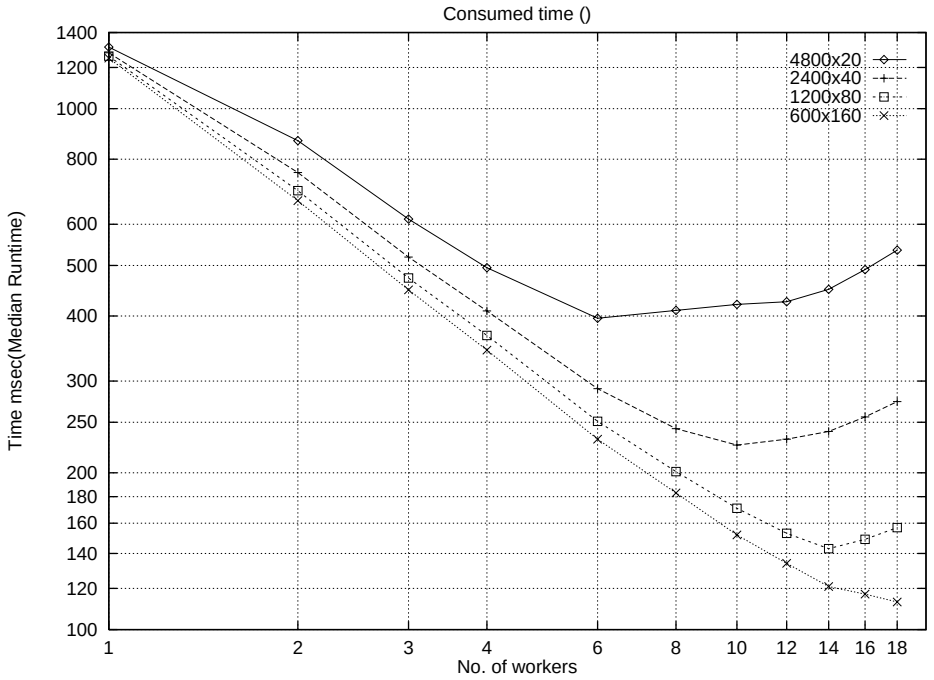

Figure 3: Matrix multiplication

Benchmark	N	T_1	C	V
4800x20	20	1313	36	0.26
2400x40	40	1280	19	0.52
1200x80	80	1261	12	1.04
600x160	160	1250	8	2.07

Table 8: Time to do one vector multiplication in milliseconds

Workers	O_2	O_3	O_4	O_6
4800x20	212	177	167	
2400x40	114	93	89	77
1200x80	66	53	52	41
600x60	41	33	32	24

Table 9: Matrix multiplication, overhead compared to ideal

Workers	2	3	4	6
$\mathbf{S_W}$	0.081	0.10	0.13	0.18
$\mathbf{B_W}$	17	12	13	6

Table 10: Matrix multiplication, scheduling overhead

The explanation to this is to find in number of scheduling operations and the size of the tasks.

3.1 Initial overhead

Given the time to create the initial matrix and vector C, and the total execution time using one worker T_1, the time to do one vector multiplication can be calculated. Table 8 shows the measured time C and calculated time V for one vector operation, for each benchmark. The time C, to create the matrix and vector, is almost directly proportional to $M + 2N$, since the same vector is used for all rows in the matrix, i.e only one vector is actually created for the matrix. The calculated time V is as expected almost directly proportional to N, the number of columns.

We then investigate the execution time when using two, three, four and six workers. Table 9 shows the total overhead O_W when more than one worker is used. The overhead is calculated as

$$O_W = T_W - (T_1/W)$$

where T_W is the median execution time using W workers. The figure for the first benchmark using six workers has been omitted since it is affected by limitations explained in the following section.

Since the overhead is partly induced by the scheduling operations it should be possible to describe it as a function of the number of rows executed by each worker.

$$O_W = (M/W)S_W + B_W$$

Given the samples of O_W in Table 9 we can estimate the values of the constants S_W and B_W. If we omit the six-worker figure for the first benchmark, we get a perfect (1.0) correlation using the values shown in Table 10.

We know that the structure of the matrix benchmark induces a scheduling operation for each of the M rows in the matrix. When one worker is used this is a local operation but when several workers are used a global scheduling operation is performed. This global scheduling operation takes S_W milliseconds. The calculated value of S_W increases as the number of workers increase. This is natural since the searching for a task in the scheduler is more complicated as the number of workers increase. The equation

$$S_W = 0.024W + 0.032$$

correlates exactly to the obtained values.

The scheduling overhead when two workers are used is thus 0.08 milliseconds but increases with 0.024 milliseconds for each worker that is added. The total execution time is given by

$$T_W = (T_1/W) + (M/W)(0.024W + 0.032) + B_W$$

where the value of B_W is found in Table 10.

Apart from the figure B_W we have a perfect description of the overhead when up to six workers are used for all but the first benchmark. The first benchmark is limited by other factors when six workers are used. The parameter B_W is complex to analyze since it is related to how the matrix is created, initial cost of running in parallel etc. The parameter constitute less than two percent of the execution time.

Even if we have successfully understood the difference in execution time when two to six workers are used there is still one question that is unanswered. Why does the execution time level-out, and why does it level-out at different levels?

3.2 The limiting factor

It is the scheduling overhead that explains the initial difference between the benchmarks but it does not explain the difference in minimal execution time. To explain this we have to understand the scheduling operation in detail.

When a worker executes the `vmul/4` procedure call it also adds a continuation task on its own stack. This task is referring to the recursive `multiply/3` procedure call and is the key to the rest of the multiplication. A worker that steals the task has to initiate the call to the vector multiplier for the next row before a new continuation task is produced. This continuation can then be stolen by another worker. Observe that there is a single continuation task in the whole system that is moving around. The time duration between the two steal operations in the system is called the turn-around time D. The turn-around time D and the time to multiply one vector V, shown in Table 8 sets a limit on how many workers that can productively take part in the execution to be roughly $\lceil (V + D)/D \rceil$. If more workers are allocated they will only wait for the continuation task to be passed around.

If we divide the obtained minimal execution time (as shown in Figure 3) with the number of rows in the matrix, we get an estimate of the turn-around time for the given benchmarks to be (0.08, 0.09, 0.12, 0.19) respectively. The figure for the last benchmark is overestimated since we did not reach the limit of the system with the available twenty processors. Following the reasoning in the previous paragraph we see that the useful number of workers to perform the benchmark increases as the granularity increases.

The smallest granularity of work that is profitable to use is thus depending on the number of workers that should productively take part in the execution. In a two processor system the granularity can be small whereas a twenty processor system need larger granularity to keep the workers busy. This is of course only applicable if the tasks are scheduled sequentially.

3.3 Remedy

In building the system it is clear that the scheduling overhead must be as low as possible however it is also possible to rewrite programs to generate larger task. For other techniques to transform sequential programs see the articles by Debray and Jain [6] and Hermenegildo and Carro [8].

```
cell(0, State, Last, This, _, _, _, _, _, _, _, _):-
        ->      Last = State,
                This = [].
cell(G, State, Last, This,
                [NW|NWr], [N|Nr], [NE|NEr],
                [W |Wr],     [E |Er],
                [SW|SWr], [S|Sr],[SE|SEr]) :-
        ->      G1 is G-1 ,
                T is NW+N+N+W+E+SW+S+SE,
                change(State, T, New),
                This = [New|Rest],
                cell(G1, New, Last, Rest,
                        NWr, Nr, NEr,
                        Wr,   Er,
                        SWr, Sr, SEr).
```

Figure 4: A cell in life

The basic problem we faced was generally due to the bottleneck in producing global tasks to get workers busy. The Penny system will not magically be able to reduce this bottleneck for the matrix program. However a small change to the program does the trick. If we represent the matrix as a tree instead of a list we can write the following matrix multiplier.

```
m_multiply(v(V0), V1, R0, R):-
        ->      R0 = [V|R],
                v_multiply(V0, V1, V).
m_multiply(m(ML,MR), V, R0, R):-
        ->      m_multiply(ML, V, R0, R1),
                m_multiply(MR, V, R1, R).
```

Written in this way the system has no difficulties in obtaining good speedup even for matrices with small rows.

4 Communicating processes

The benchmark that we examine in this section is extreme in the ratio of communication over calculation. This means that the dominating factor of the execution time does not originate from the instruction decoding but rather from the suspension and scheduling mechanism.

In the "game of life" benchmark each cell in the grid is implemented by an AKL process. The grid has the shape of a toroid so there are no borders that have to be treated specially. The program consists of two parts: the building of the toroid and the communication between cells. The building phase only takes about 5% of the execution time and shows in itself good speedup. The building phase is of course executed in parallel with the computation of the first generation. Each cell is implemented by the `cell/12` process listed in Figure 4. The first argument is the number of generations, the second and third are the current and final state, the fourth the outgoing stream of the history cell state and the remaining arguments are the incoming streams from the neighbors. The process suspends, waiting for information from all of its neighbors, and then calculate its own next state.

The smallest benchmark uses a toroid of size 30×30 and computes 10 generations. In the larger benchmarks either

Workers	2	4	8	12	16	18
10 30x30	1.90	3.10	4.66	5.38	5.69	5.62
10 60x60	1.96	3.24	4.81	5.58	5.77	5.69
10 120x120	1.97	3.29	4.87	5.64	5.82	5.68
40 30x30	1.90	3.42	5.88	7.51	8.10	8.21
160 30x30	1.89	3.54	6.15	7.95	9.00	9.19

Table 11: Speedup (excluding gc) for the game of life

Workers	2	4	8	16
10 30x30	15	7.6	4.5	3.8
10 60x60	16	6.8	4.2	3.5
10 120x120	17	6.5	4.1	2.9
40 30x30	58	23	10	8.5
160 30x30	232	53	16	11

Table 12: Ratio of executed/stolen tasks

the toroid or the number of generations is increased. The execution time is directly proportional to the number of generations and the size of the toroid. Garbage collection was invoked only in the two largest benchmarks and constituted only one percent of the total runtime.

Table 11 shows the speedup, based on the median execution time excluding garbage collection time, of twenty runs. As clearly seen the speedup is not very good, except for the two last benchmarks,

4.1 Stolen tasks

The reason for the nonlinear speedup is partly to be found in the ratio between the number of executed and stolen tasks. When one worker is used each cell is in average suspended on four of its neighbors. Since in average four of the neighbors have already calculated their generation only four neighbors remain. This means that in average four wake tasks have to be handled before the next generation can be computed. The tasks are however not identical, the first three tasks only remove a constraint whereas the fourth wakes and executes the suspended and-node, produces new wake tasks and suspends the goal of the next generation. As the number of workers increase the number executed tasks decrease but only marginally.

The time to handle one task is in average 0.07 milliseconds. If there is a 0.1 millisecond penalty to steal a task it is of course crucial that more than one task gets executed. Table 12 lists the number of executed tasks per stolen task during the execution for different numbers of workers. The ratio correlates well to the obtained speedup.

4.2 Cache performance

Running the benchmarks in SimICS gives very interesting figures on the read miss rate shown in Table 13. The miss rate increases as expected when the number of workers grow. The final miss rates of the "30 x 30" benchmarks are very high. The figures are higher than for the quick-sort benchmark and more than twice the miss rate of the turtles benchmark.

It is also interesting to notice that the miss rate is better for the benchmarks with larger grid. This indicates that the workers spread in the grid and the larger grid reduces the conflicts. Notice that the miss rate actually decreases in the "10 120x120" benchmark when going from two to four workers. This could be a consequence of having more cache memory in total when running on four processors.

Workers	2	4	8	16
10 30x30	0.20 %	0.53 %	0.93 %	1.59 %
10 60x60	0.14 %	0.32 %	0.47 %	0.91 %
10 120x120	0.25 %	0.18 %	0.42 %	0.65 %
40 30x30	0.33 %	0.59 %	1.02 %	1.36 %
160 30x30	0.41 %	0.66 %	0.99 %	1.42 %

Table 13: Life, read cache performance

How significant is the cache performance? The total number of read operations in the "160 30x30" benchmark when using two workers is about 225 million. When sixteen workers are used the total is also 225 million; the extra number of workers do not increase the number of read operations. The nonlinear speedup could of course be explained by an unbalance in the distribution of work but apart from the first worker that performs some extra work during garbage collection, the workers all perform 14 million read operation and differ only by 25 thousand operations. The first worker performs 14.9 million operations, only 6% more operations than the worker with the lowest read count. The work is thus extremely balanced and there is no overhead in the number of operations that can explain the nonlinear speedup.

In earlier experiments we have found a correlation between the execution time and the number of read operations. The following equation has a 0.96 correlation to the obtained execution time:

$$0.1 * \text{Reads} + 0.33 * \text{Read-misses}_{L1} + 10 * \text{Read-misses}_{L2}$$

We know from that about 10% of all read operations misses the first level cache. If we insert the figures for the "160 30x30" benchmark using sixteen workers, we get the figure 3650 milliseconds where the actual execution time is, including garbage collection, 3439 milliseconds. If we assume a second level read miss rate of 0.15%, which is the miss rate when running on one worker, we get an estimated execution time of 1888 milliseconds i.e. a speedup of thirteen instead of seven. The cache miss rate is thus for the "30x30" benchmarks a limiting factor.

In an experiment with a "160 120x120" benchmark the ratio of executed to stolen tasks was 23 when sixteen workers were used. The speedup, excluding garbage, collection was 12.5. The execution time for one worker was about 12 minutes. Running this benchmark in SimICS would take about a 20 hours.

5 Competing with C

We have shown that the parallel performance of the Penny system is very good but that the limiting factor is often the cache performance. How will a optimized Penny system perform? We know that we can double the initial performance of the system with a better compiler and a native code compiler will double that figure again. When the performance increase the number of read operations might be concentrated in time and limit the performance even further. Will

Workers	1	2	4	6	8	16	18
256/32	3586	1846	939	641	496	357	356
64/64	3361	1718	876	596	459	276	259
16/128	3258	1666	853	598	456	286	272
4/256	3206	1648	863	632	507	318	298
1/512	3215	1785	945	670	534	370	358

Table 14: Smith-Waterman, execution time in milliseconds

the cache performance stop us from executing at the speed of C?

To test this we selected a benchmark that originates from a practical problem. The Smith-Waterman algorithm [16] computes a value that is a measurement of how good two DNA sequences align. It can be visualized as calculating the value in the lower right corner of a matrix where each element is depending on its west, north-west and north neighbors. The uppermost row is given by one sequence and the leftmost column by the other sequence. The execution obviously has to begin in the upper left corner and can then proceed towards the lower right corner.

The program was originally written in Prolog and only small changes were needed to make it run in parallel in the Penny system. The original problem was to match one sequence of size 32 to each of one hundred sequences of equal size and to select the best. The parallelism was obvious since the matching operation are independent. In the Penny version not only the independent matches can be calculated in parallel but the matching procedure itself is used to extract parallelism.

To mimic the effect of a optimized Penny system the arithmetic operation that calculates the value of a position was implemented as a built-in. In order to do this a new data type was needed. This was easily done with the generic object interface. The new data type holds four integers and is used to represent the value of a position. Only four builtin operations were defined: a recognizer, a constructor, a selector and the arithmetic function. The resulting program is about six times faster than the pure Penny program – faster than SICStus v3 native code.

Table 14 shows the execution time of the program for different data sets N/M where N is the number of sequences in the database and M the length of the sequences. Since the algorithm is quadratic we have reduced the number of sequences with four as the length is doubled. The benchmarks performs very well for all values of N and M. The best speedup is reached in the 64×64 benchmark where the final speedup is thirteen.

The limited speedup in the 256/32 benchmark could be a result of several workers that compete inside one computation instead of selecting there own sequences.

The execution time for a well written C program compiled with `gcc -O4` is shown in table 15. The increasing numbers for the C program can only be explained with a decreased cache performance. As the sequences grow larger the data structures do not fit into the cache. This is verified by running the C program in SimICS, the cache miss rate grows from 0.02% to 0.4%. This is a penalty that is taken by the Penny system in almost all benchmarks.

The Penny program executes at about the same speed as the C program when three to six workers are used. The final execution time for the 64×64 benchmark is 259 milliseconds which is 2.5 times better than the C program. The

Benchmark	265/32	64/64	16/128	4/256	1/512
C ms.:	590	650	680	820	1330
Penny ms.:	356	259	272	298	358
C/Penny :	1.7	2.5	2.5	2.8	3.7

Table 15: Smith-Waterman, C vs. Penny using 18 processors

Workers	1	2	4	6	10	16
including output						
time (s):	30.2	17.6	14.8	12.7	12.8	12.9
speedup:	1	1.7	2.0	2.4	2.4	2.3
excluding output						
time (s)	19.2	9.78	5.24	3.91	3.91	4.00
speedup:	1	2.0	3.7	4.9	4.9	4.8

Table 16: The compiler compiling itself

1×512 benchmarks is almost four times faster then the C program. The cache performance did not severely limit the performance even though we optimized the Penny program to mimic a native code compiler.

6 The compiler

The Penny compiler can of course be executed in the Penny system itself. It consists of about 260 definitions using 1400 clauses, all in all about 3000 lines of AKL code including parser and output routines. It compiles itself in about thirty seconds using one worker. The parallel performance is initially very good but then levels out at about thirteen seconds when six workers are used. A total speedup of little more than two is of course nothing to be proud of - what is happening?

If we look at the number of stolen tasks and the average execution time for each stolen task the figures do not indicate that the number of scheduling operations would cause a problem. The scheduling operations are quite few and the average execution time for each stolen task is quite high. The average idle time is however very high. This indicates that the workers have a hard time finding available work.

The amount of parallelism during the execution is however not constant. In Figure 5 the average number of workers that are actually busy (in each 500 ms. intervals) during the execution is shown for three runs with 2, 4 and 6 workers. As clearly seen the parallelism is very high in the initial phase of the execution but after a couple of second only one worker is active.

The reason is found in the output procedure. This procedure is completely sequential and cannot be parallelized. Moreover the output routine is very poorly optimized, it is coded in AKL down to the output of atoms, strings and individual characters. When the output routine is excluded the execution time and speedup is, as shown in Table 16, improved.

Given the figures with the output routine excluded we see that the routine was responsible for eleven seconds of the total execution time. Given that this sets a limit on the minimal execution time it is not that bad when the actual minimal execution time is 12.7. It does however not explain why the minimal execution time is not reached al-

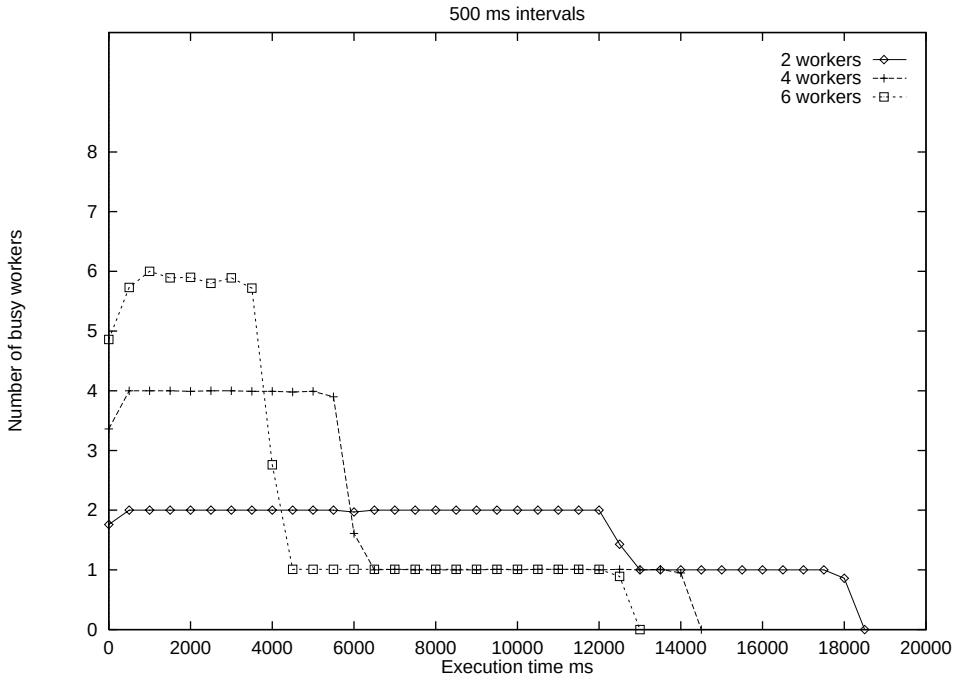

Figure 5: Number of busy workers during an execution

ready when three workers are used. If two workers can do the actual compilation in less than ten seconds (the first definition is compiled in a couple of milliseconds) there is no reason why the output routine should not be completed by a third worker in much the same time.

The output routine must, to minimize the execution time be scheduled as soon as data is available. The scheduler has of course no knowledge of this and schedules work that is not critical i.e. the compilation. All workers could be assigned to do the compilation while no one is scheduled to the output routine.

We are actually lucky that the output routine is scheduled at all, since the compilation phase can keep six workers busy. In a six worker run, all six workers could be assigned to the compilation and only when this work was exhausted would the output routine be scheduled. This would mean that the six worker run would complete the whole execution in $3.9 + 11.0 = 14.9$ seconds instead of the actual 12.7 seconds.

The good thing is that the output routine is in practice scheduled. The bad things is that if it was not, there would be very little we could do about it. By making parallelism transparent to the programmer we have also removed all possibilities to control it. This is a big drawback if one sees the system as a parallel programming language.

7 The enterprise

The Penny system was developed on a SPARCcenter2000. A machine that has been a good representative for modern cache-based shared-memory architecture. This changed last year when Sun released their Enterprise series of Ultra based machines. The 167MHz UltraSPARC processor is about three times faster than the 50MHz SuperSPARC processor that are found in the SPARCcenter2000. The question is how this affects the cache performance of the system. We have shown that cache performance can be one of the limiting factors and faster processors can make the situation even worse.

Workers	2	4	8
matrix	1.97	3.66	7.46
hanoi	2.00	3.86	7.97
fib	2.00	3.91	7.72
tak	1.98	3.60	7.73
mastermind	1.79	3.59	7.42
kqueens	1.97	3.77	7.47
turtles	1.90	3.29	6.20
qsort	1.90	3.55	6.43
160 30x30	1.93	3.35	6.41
1/512	2.01	3.75	6.79

Table 17: Speedup on Enterprise 167 MHz

Preliminary results show that the Penny system performs even better on the Enterprise. Table 7 show the speedup based on the median execution time of eleven runs. The same Penny binary was used; no changes were made for the UltraSPARC 64bit architecture. One possible reason for the improved figures is that the buss architecture on the Enterprise is more than three times better than the bus architecture on the SPARCcenter. The reason for the limited performance of the SPARCcenter could be an indication on that the bus is saturated. The bus architecture on the Enterprise is much better and this seams not to be problem.

8 Conclusions

An implicit parallel system relieves the programmer from the burden of explicitly having to deal with parallelism. On the other hand the programmer has little control over the parallel execution. In a fine-grain system the overhead for

managing the parallelism can become larger than the gain of running on more processors. If this is the case there is little the programmer can do about it.

The limiting factor in a program is first of all sequential threads that can not be broken up into smaller task. The limit in speedup is then just a consequence of Amdahl's law. Although there are no control primitives in the Penny system it manages to extract the parallelism but there are of course no guarantees that system finds the optimal solution. It is however not easy even in an explicit parallel system to obtain the best possible speedup.

In programs that divide up into more or less independent parts there is little gain in doing an explicit allocations of goals to processors. The Penny system does a very good job in managing the tasks. The stream and-parallel programs are mainly limited by their cache performance and this is not solved by explicit allocation or granularity control.

Implicit parallelism also works well in programs with very fine-grain tasks. The scheduling overhead is about 0.1 milliseconds and this does set a limit on the obtainable speedup but even a program with a task size of 0.26 millisecond shows a speedup of 3.3 on six processors. A sequential thread in the scheduling operation is a more serious threat. The turn-around time between two scheduling operations limits both the execution time and the number of workers that can productively take part in an execution.

Our conclusion is that implicit parallelism works very well but good cache performance can not be overestimated. In a parallel system it is vital that miss rate of the second level cache is kept to a minimum. The Penny system has been designed with cache performance in consideration. All structures that are used in the execution state use their own cache lines to avoid false sharing. Since the AKL terms are use in a program to communicate between AKL processes they are often produced by one worker and later read by another worker. It is therefore very hard to minimize this source of cache misses.

Acknowledgments The parallel implementation of AKL has been developed using the AGENTS 1.0 [10] system as a starting point. Haruyasu Ueda did much of the implementation and analysis of the scheduler. Galal Atlam and Khayri Ali, designed and implemented the garbage collector [3].

Thanks to Peter Fritzson at Linköping University for access to a 20-processor SC2000 for the Penny timings.

Various parts of this work have been sponsored the European Commission in the ACCLAIM Esprit project, EP 7195 and SICS.

References

[1] Ali, K. A. M. A parallel copying garbage collection scheme for shared-memory multiprocessors. *New Generation Computing 13*, 4 (December 1995).

[2] Ali, K. A. M., and Karlsson, R. The MUSE or-parallel Prolog model and its performance. In *North American Conference on Logic Programming* (October 1990), MIT Press.

[3] Atlam, G. A. M. A. *Parallel Garbage Collection in a Multiprocessors Implementation of a Concurrent Constraint Programming System.* Phd thesis, Menoufia University, Egypt, Jan. 1997.

[4] Catanzaro, B. *Multiprocessor System Architecture.* SunSoft Press, 1994.

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

[6] Debray, S., and Jain, M. A simple program transformation for parallelism. In *Proceedings of the 1994 International Logic Programming Symposium* (Ithaca, 1994), M. Bruynooghe, Ed., ALP, MIT Press.

[7] Fanzén, T., Hairidi, S., and Jansson, S. An overview of AKL. In *ELP'91 Extensions of Logic Programming* (1992), no. 596 in LNAI, Springer-Verlag.

[8] Hermenegildo, M., and Carro, M. Relating data-parallelism and (and-) parallelism in logic programs. In *Lecture Notes in Computer Science, 966* (August 1995), Springer-Verlag, pp. 27–41.

[9] Janson, S. AKL: A multiparadigm programming language. Uppsala Thesis in Computing Science 19, SICS Dissertation Series 14, Uppsala University, SICS, 1994.

[10] Janson, S., and Montelius, J. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[11] Janson, S., Montelius, J., and Haridi, S. *Ports for Objects in Concurrent Logic Programs.* MIT Press, 1993, ch. 8, pp. 211–231.

[12] Lewis, B., and Berg, D. J. *Threads Primer.* SunSoft Press, 1996.

[13] Magnusson, P., and Werner, B. Efficient Memory Simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium* (1995).

[14] Mohr, E., Kranz, D., and Halstead, R. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* (90).

[15] Montelius, J., and Ali, K. A. M. An and/or-parallel implementation of AKL. *New Generation Computing 13*, 4 (December 1995).

[16] Smith, T. F., and Waterman, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology 147* (1981), 195–197.

[17] Warren, D. H. D. An abstract Prolog instruction set. Tech. Rep. 309, SRI International, 1983.