ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Benchmark Library

Deliverable number:	D10
Workpackage:	Resource-Oriented Specialization (WP4)
Preparation date:	1 May 2004
Due date:	1 May 2004
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Univ. of Southampton

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

In this deliverable we systematically study the influence of various factors on the performance of Prolog programs. For this we have developed a benchmark library, along with a platform independent benchmarking program. This program can be used to obtain the performance characteristics of a particular hardware platform and Prolog compiler.

Contents

1 Introduction

Software development is a costly task. This situation is being alleviated lately by software *reuse*, which allows the construction of software systems from already written and tested software. However, they tend to generate software systems which are not optimal with respect to the computing resources they need. The benefits of these approaches are thus seldom available in the realm of pervasive computing. In this context, there is a large number of computing devices which may range from personal computers to PDAs, mobile phones, dedicated processors, smart cards, wearable computers and such like. These devices have a wide range of performance characteristics; often having a relatively small amount of computing resources [6]. Time efficiency is an issue since often these devices have to operate on real-time tasks. Also, and possibly more importantly, memory efficiency is an issue. If either the software used is too large to fit in the device or needs too much memory to run, then it is simply not possible to use such software.

1.1 Resource Aware Specialisation

Program specialisation aims at improving the overall performance of programs by performing source to source program transformations. A common approach evaluation is to exploit partial knowledge about the input. Program specialisation techniques for logic programming languages have advanced steadily over the past 20 years. Algorithms have been developed which increase the amount of specialisation achieved, while still ensuring termination of the specialisation process.

However, existing specialisation systems do not use a sufficiently precise model of the compiler of the target system to guide their decisions during specialisation. In addition to execution speed, there are many other important factors which are neglected by current specialisation techniques such as *partial evaluation*. Some of these factors are: size of the resulting (specialized) program, memory usage, and low-level implementation issues. While there is some recent work [3] to address this, it is a largely ignored area and some of the problematic issues raised in [5] are still valid today.

A suitable *low-level cost model* would allow a partial deduction system to make more informed choices about the local control (e.g., is this unfolding step going to be detrimental to performance) and global control (e.g., does this extra polyvariance really pay off). Some promising initial work on cost models for logic and functional programming has already been made in [1, 2]. However, such a low-level cost model will depend on both the particular Prolog compiler and on the target architecture and it is hence unlikely that one can find an appropriate mathematical theory. This means that further progress on the control of partial deduction will

probably not come from ever more refined mathematical techniques such as new well-quasi orders, but probably more from heuristics and *artificial intelligence* techniques such as case-based reasoning or machine learning. For example, one might imagine a *self-tuning* system, which derives its own cost model of the particular compiler and architecture by trial and error. Such an approach has already proven to be highly successful in the context of optimising scientific linear algebra software [7].

1.2 Aim

In this deliverable we systematically study the influence of various factors on the performance of Prolog programs. For this we have developed a benchmark library, along with a platform independent benchmarking program. This program can be used to obtain the performance characteristics of a particular hardware platform and Prolog compiler.

There are several possible uses of the results obtained in this deliverable. First, the results indeed show that hardware and Prolog system have a considerable influence on the performance of Prolog programs, which Prolog specialisers and optimisers should be aware of. Second, our results and tools should help us to derive a realistic low-level, empirical cost-model that can be used to guide the next generation program specialisers. Finally, our program can also be used to compare new Prolog systems or new hardware systems, and identify unexpected behaviour (e.g., a "bug" in the Ciao compiler was spotted by our benchmarking tool).

2 Building up the Perfomance Model

The first step in the assessment of the performance of a program that will be fitted to a device with limited resources, is to break the measurement into simple tasks. We call these tasks: *atomic parameters*. These parameters are the basis for empirically getting an idea of the various of specialising, when this specialisation is a non-deterministic process. The following atomic parameters can be identified in a Prolog program.

Unification:

Unification is at the basis of Prolog execution, and hence it is important to get a good idea of its cost when modelling the performance of a system. The idea behind unification, which was borrowed from automated theorem proving [4], is that two terms match if you can instantiate their variables to values in such a way that the two expressions become identical. This binary operation would attempt to make its two operands the same, however complex the data object

is. Thus, in our experiments we refer to two types of unification: simple and deep. By simple unification an arbitrary list of unbounded variables gets unified to another list of the same length:

$$[X_1, X_2, \ldots, X_n] = [Y_1, Y_2, \ldots, Y_n]$$

On the other hand, we also examine the unification of more complex terms, where the binding is *deeply* wrapped by compound terms:

$$f_1\left(f_2\left(\cdots f_m\left(X_1, X_2, \ldots, X_n\right)\cdots\right)\right) = f_1\left(f_2\left(\cdots f_m\left(Y_1, Y_2, \ldots, Y_n\right)\cdots\right)\right)$$

These two types of unification are tagged `unification/n` and `deep_unif/n`, where `n` is the length of the list in the first case and the *arity* of the predicate in the latter one.

Accessibility:

In logic programming, the clauses of a predicate can be defined either statically, dynamically, or even accessed from a mass storage device. The time to validate a query depends on this factor. We use the following definitions:

$$\begin{aligned} P_S &= \{p(X_1, X_2, \ldots, X_n) \leftarrow \ \forall\, 0 \leq i < n,\, 0 \leq X_i < m\} \\ P_D &= \{\text{assert}(p(X_1, X_2, \ldots, X_n)) \leftarrow \ \forall\, 0 \leq i < n,\, 0 \leq X_i < m\} \end{aligned}$$

and then the goal $G =\leftarrow\ p(C, C, \ldots, C), C\ =\ m/2$ is executed for each program: $P_S \cup \{G\}$, $P_D \cup \{G\}$ to test the difference between a predicate that is defined statically and a dynamic one.

$$P_{io} \ = \ \{\text{read}(\text{Stream}, P_S) \leftarrow \ \forall\, 0 \leq i < n,\, 0 \leq X_i < m\}$$

The above definition is used at a later stage to measure the time it takes to read a set of clauses from a mass storage device.

Coroutining:

Coroutines passively evaluate consistency conditions, as opposed to the traditional approach in declarative languages. Although they are inadequate as a general constraint mechanism, coroutining allows the designer to define more complex constraints. In our benchmarking procedure, the `call_residue/2` and `when/2` built-ins are tested in order to analyse the overhead introduced by coroutining. These built-ins are used to wrap a call to another procedure, discarding the coroutining relevant information – we are only interested in measuring the overhead. Thus, we have:

```
call_residue(X,_).
when(ground(a),X).
```

where X is bound to either a simple unification, as described above, or to a more complex *naive reverse* example to test scalability. Again, the arity used in the declaration of this test will be passed around to X.

Arithmetic:

Differentiating between the complexity associated to integer or floating-point arithmetic is desired in most empirical models. We run the following test:

$$Y = \sum_{i=1}^{n} \prod_{j=1}^{m} X_{i,j}$$

where $m, n \in \mathbb{N}$ are the number of variables in each term, and the number of terms respectively. This is being implemented by `sum_prod/3`, which is then used in the following four benchmarks: `non_float/n` for $X_{i,j} \in \mathbb{N}$, `float/n` for $X_{i,j} \in \mathbb{R}$, `small_int/n` and `small_float/n` when $X_{i,j} \approx 0$.

2.1 Multi-Platform Execution

In order to compare figures, we need to identify which specifications are in use at the time we run the benchmarks. Firstly, we can identify the Prolog by an internal tag used in most systems.

```
which_prolog(ciao) :-
    current_prolog_flag(version,Info), nonvar(Info),
    Info = ciao(_,_),!.
which_prolog(sicstus) :-
    current_prolog_flag(version,Info), atom(Info),
    sub_atom(Info,0,7,_,'SICStus'),!.
which_prolog(swi) :-
    (current_prolog_flag(unix,_);
    current_prolog_flag(windows,_)),!.
which_prolog(xsb) :-
    xsb_configuration(engine_mode,'slg-wam'),!.
```

Some Prolog languages have significant differences throughout their development, while others have a less active developing. Identifying the version number could be of interest in those very dynamic Prologs.

```
which_version(Version) :-
    current_prolog_flag(version,Info), nonvar(Info),
    Info = ciao(Version,_),!.
which_version(Version) :-
    current_prolog_flag(version,Info), atom(Info),
    pick_using(' ',Info,Version),!.
which_version(Version) :-
    current_prolog_flag(executable,Info), atom(Info),
    pick_using('-',Info,Ver),
    prefix_using('/',Ver,Version),!.
which_version(Version) :-
    xsb_configuration(version,Version),!.
```

Based on this information, we can identify the operating system as well as the architecture as follows.

```
which_os(OS) :- which_prolog(ciao),!,get_os(OS).
which_os(OS) :-
    current_prolog_flag(host_type,Info), atom(Info),
    pick_using('-',Info,OS),!.
which_os(OS) :-
    current_prolog_flag(arch,Info), atom(Info),
    pick_using('-',Info,Sys),
    pure_atom(Sys,OS),!.
which_os(OS) :-
    xsb_configuration(os_type,OS),!.

which_arch(Arch) :- which_prolog(ciao),!,get_arch(Arch).
which_arch(Arch) :-
    current_prolog_flag(host_type,Info), atom(Info),
    prefix_using('-',Info,Arch),!.
which_arch(Arch) :-
    current_prolog_flag(arch,Info), atom(Info),
    prefix_using('-',Info,Arch),!.
which_arch(Arch) :-
    xsb_configuration(host_cpu,Arch),!.
```

These predicates are used to obtain information about the system. By passing an ungrounded variable as argument, e.g. `which_arch(X)`, knowing that `X` will be unified to an atom[1] that represents the architecture, e.g. `i86`. A grounded argument could be used for language-specific built-ins. For example:

```
( which_prolog(xsb) -> table p/2 ; true )
```

is used to perform tabling on `p/2` only when the program is executed in XSB Prolog.

3 Benchmarking

Evaluating the performance of devices with restrictive computational and storage capabilities is an extremely complex matter, and it is an unrealistic assumption to expect any set of benchmarks to tell the whole story. In fact, such a performance measurement would depend on so many factors that many people have justifiably questioned the usefulness of benchmarking. However, we believe that useful information can be retrieved from the benchmarking process, and that building an empirical model from these results is advantageous.

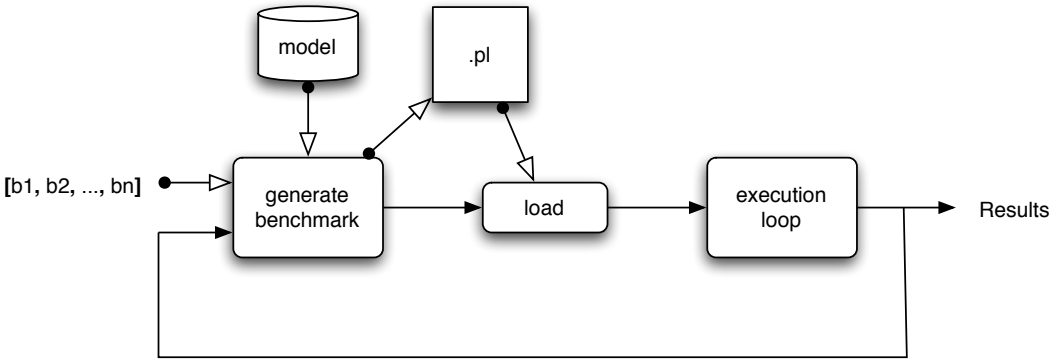

Figure 1: Structure of the algorithm

The main part of our benchmarking algorithm is divided into three processes, as shown in Figure 1. This has been coded in Prolog by the predicate `benchmark/1`, where the argument is a list `[b1/a1,b2/a2,...,bn/an]` = `[H|T]` of experiments to be executed. For each element of the list, b_i is an atom that represents the experiment and $a_i \in \mathbb{N}$ is the arity involved in the experimentation. For instance, `b1/a1` = `unification/10` refers to performing a

[1]Note that these atoms are not unique. For instance, `which_arch/1` will unify the argument to `powerpc` on SICStus and `ppc` on Ciao, and both cases refer to the same platform: a PowerPC processor.

unification of two lists, each of them with a length of 10 elements. In terms of its implementation, an entry for b_i is looked up on a *model* library. This entry defines the semantics associated with b_i.

```
benchmark([]).
benchmark([H|T]) :-
    construct_name(H,Name,TmpFile),
    which_prolog(Prolog),
    flags(Prolog,Flags),
    map(H,Exported), H = [HH|_],
    open(TmpFile,write,In),
    write_headings(In,[Name,Exported,Flags,Prolog,HH]),
    generate_benchmark(In,H),
    close(In),
    load_and_execution_loop(H,Name),
    benchmark(T).
```

Following the execution of `write_heading/2` and `generate_benchmark/2`, a temporary file is generated for each experiment b_i. The `write_heading/2` predicate provides the right declarations, according to the Prolog being used. Next, `generate_benchmark/2` creates a the main structure on which the measurement is based on. In order to obtain some meaningful timings, the benchmarking process has to be repeated many times within the measurement. This repetition, however, increases the overhead of simple actions, such as `call/1`. Thus, the temporary file created resembles a structure that avoids this overhead by means of three predicates: `repeat/2`, `call_once/0`, and `do_benchmark/2`.

```
repeat(A,A).
repeat(A,B) :-
    A>1,
    C is A-1,
    repeat(C,B).
```

```
do_benchmark(A) :-
    repeat(A,_),
    call_once,
    fail.
do_benchmark(_).
```

As opposed to naively doing a recursive loop, where the body is repeatedly called (`A` times) and the measurements are significantly affected by the overhead of `call/1`, the structure shown above does not incur in a big overhead. As `do_benchmark/1` runs it will fail and backtrack to another execution of `repeat/2`, until `A>1` is not satisfied any more. On backtracking the variable bindings are undone, leading to less memory overhead.

Due to the varied nature of the benchmarks, it is not practicable to run a number of times (A) each experiment and then compare the timing information. We propose that, having generated the temporary file, all benchmarks run as much as they need to overcome a threshold. But we keep this limit constant and measure number of runs instead. In this way, results are normalised with respect to this constant time, allowing a fairer comparison.

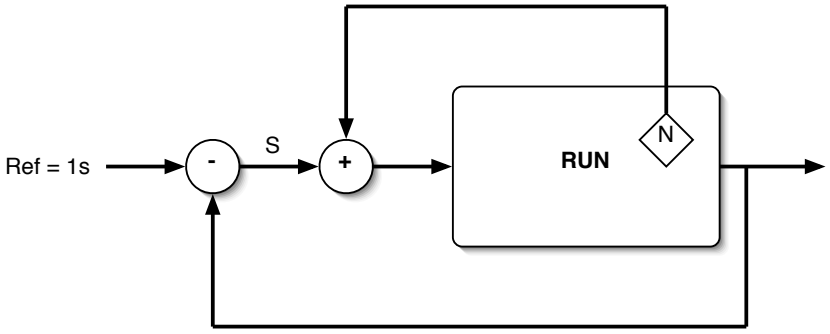

Figure 2: Execution loop

Figure 2 depicts the implementation of `execution_loop/4`, showing a double nested loop that is committed to level up the number of executions, until the reference (threshold) of 1 sec is achieved. This loop is affected by the internal variable `N`, which stores the number of executions that have taken place in the last iteration. In this way, when the measured time (`Time`) is still far from *Ref = 1s*, i.e. `Diff` is large, we get greater increments among iterations than when `Diff` is small. The Prolog code for this is:

```
execution_loop(Param,_,N,Time) :-
    ground(Time), Time >= 1000, !,
    retractall(repetitions(_)),
    assert(repetitions(N)),
    report(Param,N,Time).
execution_loop(read/Arity,Module,_,_) :-
    repetitions(N), Bench_Call =.. [bench_read,T,N],
    call(Module:Bench_Call),
    which_prolog(P),
    inform(P,read/Arity,T,N),
    report(read/Arity,N,T).
execution_loop(Param/Arity,Module,N,Time) :-
    ground(Time), Diff is 1000 - Time, Diff > 0, !,
```

```
step(S), N1 is N + S * Diff,
atom_concat(bench_,Param,BenchCall),
Bench_Call =.. [BenchCall,T,N1],
call(Module:Bench_Call),
which_prolog(P),
inform(P,Param/Arity,T,N1),
execution_loop(Param/Arity,Module,N1,T).
```

4 Language-dependant issues

The benchmarking methodology described in the previous section reports a number of results measured in *kilo-instructions/second*. As both platforms and languages were varied during the tests, it is important to differentiate between issues that concern the language in use and issues that relate to the platform itself. In this section we compare 4 different Prologs running on the same platform: a PowerPC G4 processor, Mac OS X 10.3, 512 Mb RAM. Firstly, our benchmark set has been tested in **SICStus v.3.11.1**. Secondly, we wanted to devise the overhead of running **SICStus v.3.12.0** in the **Virtual PC 7.0.1** emulator. Thirdly, we tested **Ciao 1.11 #221** and, fourthly, **SWI 5.2.3**.

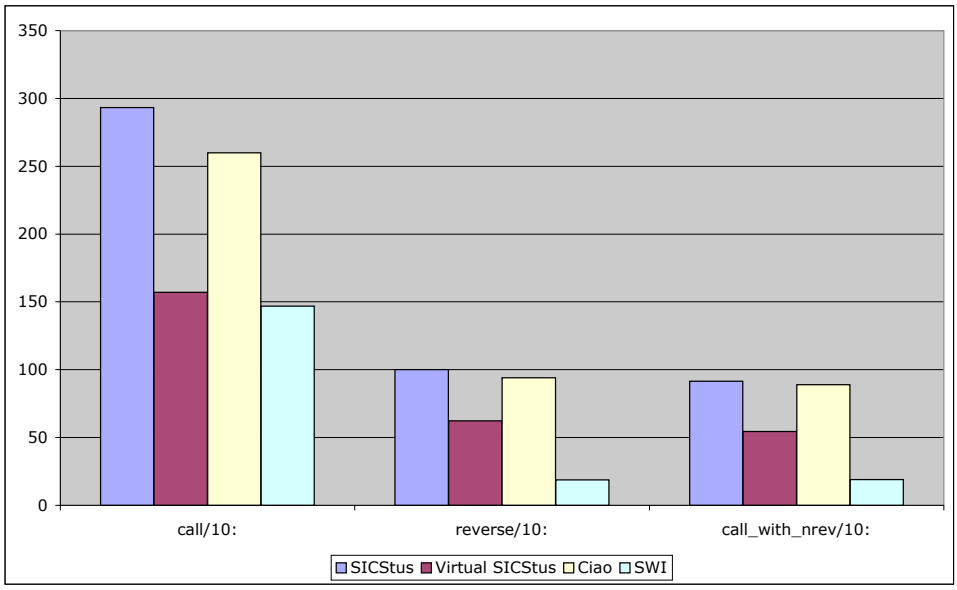

Figure 3: Same platform, different Prologs (`call`-related functions)

It can be observed from Figure 3 that the SICStus and Ciao Prologs clearly outperform SWI.

This difference becomes even more evident when the complexity of the predicate increases, as we can see from comparing the set of experiments with the other two. Despite the big overhead introduced by the emulator, it was still faster to run SICStus on *Virtual PC* than SWI.

Despite that, Figure 4 depicts how SWI remarkably surpasses both SICStus and Ciao in I/O access efficiency. Additionally, the best arithmetic (both integer and floating point) performance was obtained in Ciao Prolog.

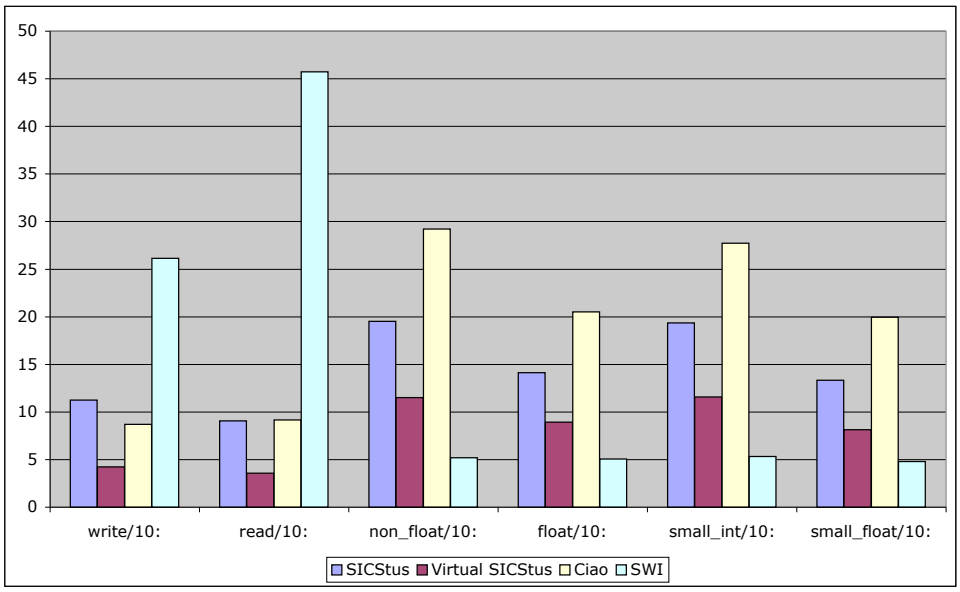

Figure 4: Same platform, different Prologs (I/O and arithmetics)

5 Platform-dependant issues

Clearly, differences on the performance of an application due to the language in which it has been implemented, does not normally overturn a designer into a different choice. In this section, we examine the effect of running our benchmarks in several platforms.

Exploring the consequences of benchmarking in several platforms could lead to wrong interpretation of the results, due to significant performance differences. For instance, we do not want to come up with the conjecture that a 2GHz processor is faster than its 1GHz counterpart, but are rather interested in the benefit than an extra 512Mb of memory would make. This is why we normalise all the results in this section, to the first test running on a PowerPC G4 processor, Mac OS X 10.3, 512 Mb RAM.

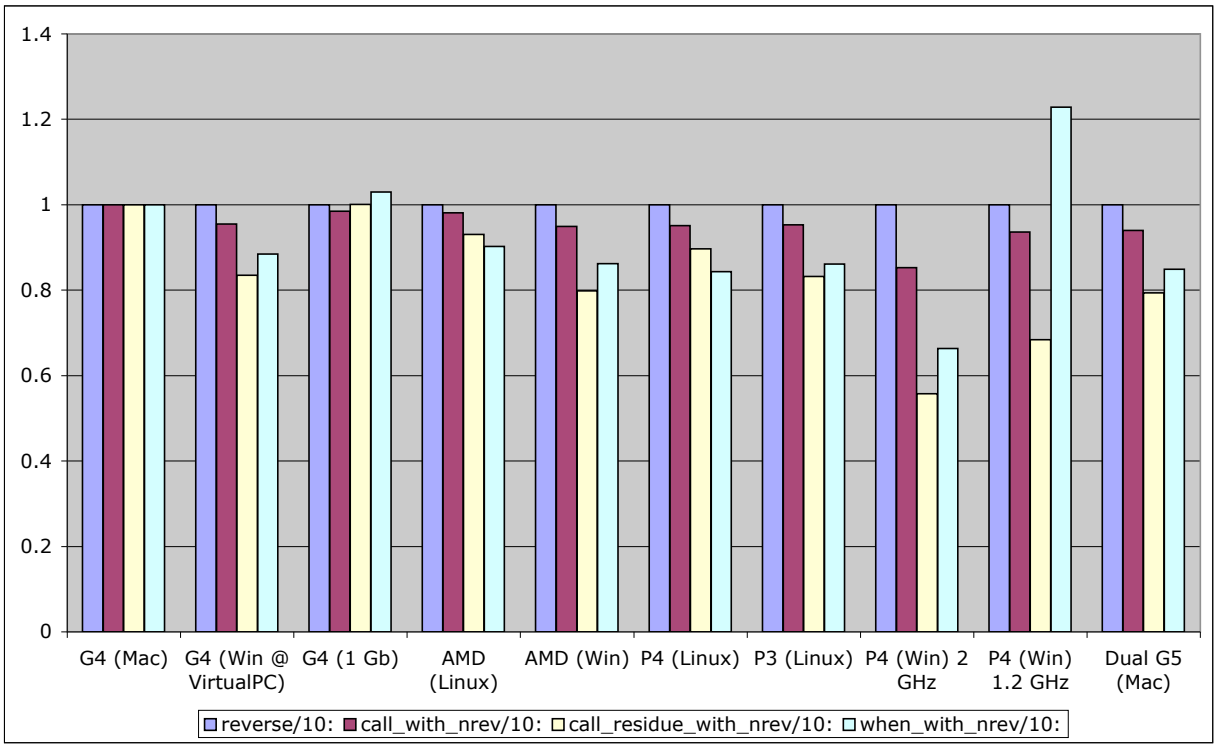

Figure 5: Naive reverse and coroutining

Figure 5 shows that there are two issues that add on to the complexity of the *naive reverse* algorithm, when using coroutining. Firstly, as it could be expected, there is the amount of memory in the system. It can easily be observed that there is less slow down due to coroutining where the system has more memory. Secondly, the operating system also plays an important role. Figure 5 shows that there is a significant added complexity by running the same experiments in exactly the same platform (AMD), but using Windows instead of Linux.

In Figure 6 we run the sum of a set of products across different platforms. The core of the benchmark consists of two recursive predicates that iterate a number of times.

```
prod_list([], Ini) :- initial_value(Ini).
prod_list([A|B], C) :-
    prod_list(B, D),
    C is A*D.

sum_prod(_, [], Ini) :- initial_value(Ini).
sum_prod(A, [B|C], D) :-
    prod_list(A, B),
```

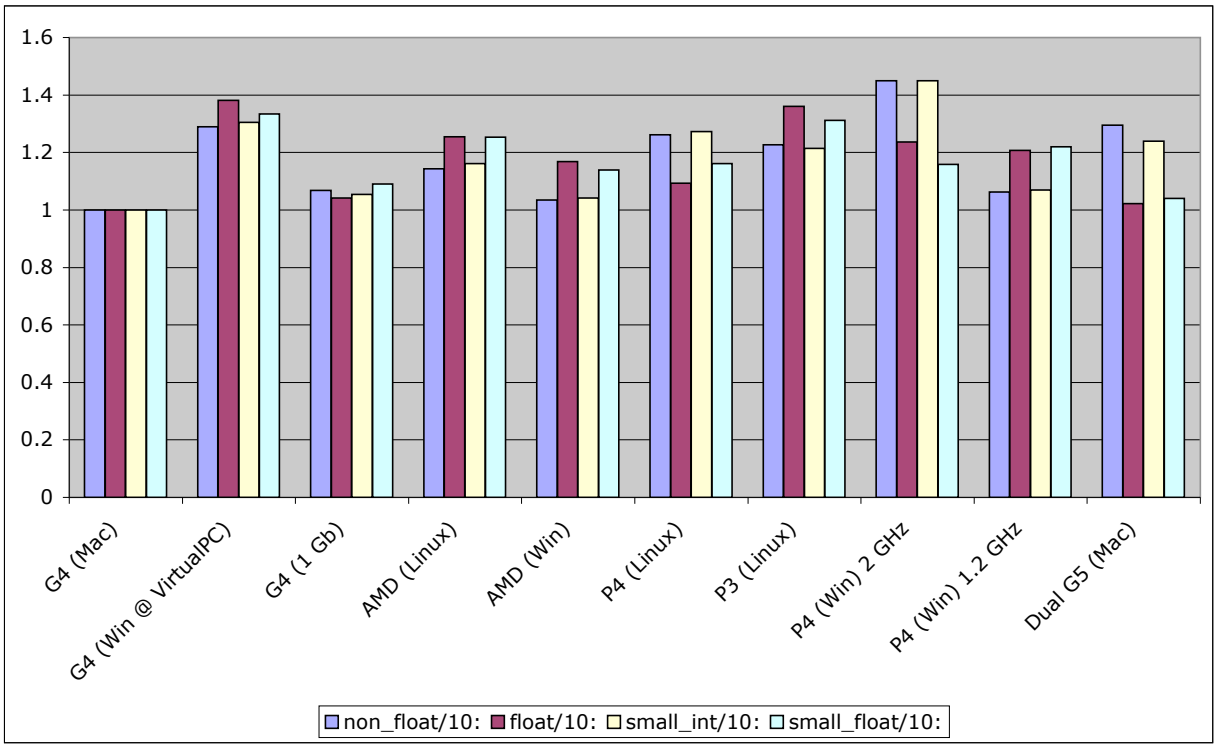

Figure 6: Arithmetic operations

```
sum_prod(A, C, E),
D is B+E.
```

To select whether we want to run this experiment using integer or floating-point arithmetics, the `initial_value` binds the variable `Ini` to one of the following values:

	int	**float**
multiplication	1	1.0
sum	0	0.0

6 Conclusion

We have presented an empirical approach to the performance model in resource-aware specialisation. Our framework is based on individually benchmarking those parameters that affect the specialisation, and reasoning from the results obtained. Important resource-awareness aspects have been analysed. We include in this deliverable extensive experimentation showing a broad comparison among languages and platforms used in the specialisation process. We believe that

our methodology can lead to an improvement in the way program specialisers work, especially in resource-aware applications, by introducing additional heuristics to the process.

References

[1] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'2000)*, pages 103–124. Springer LNCS 2042, 2001.

[2] E. Albert and G. Vidal. Source-Level Abstract Profiling for Multi-Paradigm Declarative Programs. In *Proc. of 11th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'2001*, 2001.

[3] S. Debray. Resource-bounded partial evaluation. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192, Amsterdam, The Netherlands, 1997. ACM Press.

[4] J. A. Robinson. A Machine Oriented Logic based on the Resolution Principle. *Journal of the ACM*, 10:163–174, 1963.

[5] R. Venken and B. Demoen. A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[6] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991.

[7] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.

A Empirical Results

The following results were obtained using an arity of 10. The measurement unit is *kilo instruction / seconds*.

	G4 (Mac)	P4 (Linux)
unification/10	1450.53	3115.62
deep_unif/10	1446.32	3065.53
lookup/10	2074.23	5397.99
assert/10	39.84	124.79
call/10	259.79	719.65
when/10	176.7	403.77
reverse/10	94.06	235.45
call_with_nrev/10	89.06	241.28
when_with_nrev/10	80.61	187.29
write/10	8.71	14.9
read/10	9.15	12.03
non_float/10	29.21	65.52
float/10	20.53	47.58
small_int/10	27.72	61.13
small_float/10	19.96	24.51

Table 1: Benchmarks in Ciao Prolog

	G4 (Mac)	G4 (Win @ VirtualPC)	G4 (1 Gb)	Dual G5 (Mac)
unification/10	2297.92	1052.4	2422.11	6266.83
deep_unif/10	2284.21	995.19	2602.08	6415.05
lookup/10	3079.8	1490.27	3531.58	9764.85
call/10	293.14	157.04	314.43	801.47
when/10	150.5	73.64	166.33	345.62
reverse/10	100	62.41	111	306.12
call_with_nrev/10	91.5	54.51	100	263.16
assert/10	121.9	53.91	132.04	304.88
call_residue/10	38.02	19.16	40.1	85.71
call_residue_with_nrev/10	38.23	19.92	42.48	92.94
when_with_nrev/10	69.7	38.47	79.69	181.16
write/10	11.25	4.23	11.83	20.97
read/10	9.07	3.58	10.05	20.87
non_float/10	19.53	11.53	22	69
float/10	14.14	8.95	15.52	39.41
small_int/10	19.37	11.57	21.52	65.45
small_float/10	13.34	8.15	15.33	37.82

Table 2: PowerPC Benchmarking in SICStus Prolog

	AMD (Linux)	P4 (Linux)	P3 (Linux)
unification/10	1772.5	5322.12	2185
deep_unif/10	1767.5	4562.5	2172.5
lookup/10	2557.5	8190.59	3722.5
call/10	225.23	571.08	295
when/10	103.73	252.43	128.21
reverse/10	91.58	233.64	114.68
call_with_nrev/10	82.24	203.25	100
assert/10	94.7	235.85	102.88
call_residue/10	27.65	72.05	32.85
call_residue_with_nrev/10	32.59	80.13	36.5
when_with_nrev/10	57.6	137.36	68.87
write/10	6.63	14.93	7.22
read/10	4.74	11.63	5.77
non_float/10	17.22	57.08	22.79
float/10	13.68	35.82	18.29
small_int/10	17.35	57.08	22.36
small_float/10	12.9	35.87	16.64

Table 3: Benchmarks in SICStus Prolog for Linux

	AMD (Win)	P4 (Win) 2 GHz	P4 (Win) 1.2 GHz
unification/10	2900.1	4003.83	3445.25
deep_unif/10	2843.16	4202.77	3441.5
lookup/10	4785.68	8436.56	5488.5
call/10	317.34	501.73	379
when/10	144.34	199.68	156.84
reverse/10	130.68	285.68	158.43
call_with_nrev/10	113.48	222.82	135.65
assert/10	122.37	189.11	131.1
call_residue/10	33.51	47.56	34.56
call_residue_with_nrev/10	39.88	60.9	41.45
when_with_nrev/10	78.49	132.07	135.65
write/10	9.08	12.07	9.96
read/10	7.53	10.5	8.98
non_float/10	25.5	49.33	31.13
float/10	20.85	30.45	25.6
small_int/10	25.47	48.94	31.07
small_float/10	19.17	26.93	24.39

Table 4: Benchmarks in SICStus Prolog for Windows

	G4 (Mac)	AMD (Win)	P4 (Win) 2 GHz	P4 (Win) 1.2 GHz
unification/10	160.78	165.07	443.03	203.2
deep_unif/10	178.22	179.94	404.89	201.5
assert/10	132.47	116.92	233.27	121.19
lookup/10	409.38	420.94	668.48	455.22
call/10	146.94	143.35	258.53	155.21
reverse/10	18.82	18.22	30.53	20.45
call_with_nrev/10	19	17.55	30.73	19.75
write/10	26.13	31.02	46.88	35.77
read/10	45.73	47.78	65.7	49.6
non_float/10	5.2	5.32	8.2	5.81
float/10	5.07	5.3	8.16	5.84
small_int/10	5.34	5.3	8.25	6.02
small_float/10	4.79	4.92	7.38	5.5

Table 5: Benchmarks in SWI Prolog