SICS Technical Report
T2000:01

ISRN :
SICS-T–2001/01-SE
ISSN : 1100-3154

Optimizing the SICStus Prolog virtual machine instruction set

Henrik Nässén
henrikn@sics.se

March 2001

Computer Science Department, School of Engineering
Uppsala University

Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, S164 29 Kista, Sweden

Abstract

The Swedish Institute of Computer Science (SICS) is the vendor of SICStus Prolog. To decrease execution time and reduce space requirements, variants of SICStus Prolog's virtual instruction set were investigated. Semi-automatic ways of finding candidate sets of instructions to combine or specialize were developed and used. Several virtual machines were implemented and the relationship between improvements by combinations and by specializations were investigated. The benefits of specializations and combinations of instructions to the performance of the emulator is on the average of the order of 10%. The code size reduction is 15%.

Keywords: Virtual machines and interpretation techniques, byte-code emulators, WAM, Prolog, SICStus.

Contents

1 Introduction

SICStus Prolog is one of Swedish Institute of Computer Science's (SICS's) Prolog systems. To improve execution speed and minimize space usage the virtual instruction set was investigated and modified. A methodology for finding instruction candidates for optimizations and a framework for semi-automatic testing to evaluate their impact were constructed.

The project was done as a Master of Science thesis at the Computer Science Department (CSD) at Uppsala University for the Swedish Institute of Computer Science (SICS) in Uppsala, Sweden.

The thesis is organized as follows. It first describes the history of Prolog and the basics of the WAM (Warren Abstract Machine). The layout of the tests and the various techniques that can be used to improve an emulator are discussed in Chapter 4 and 5. In Chapter 6 and thereafter follows a concrete description about how the problems formulated (first paragraph) were solved. The final chapters discuss the results and try to see into the future.

Three appendices contain additional information. Appendix A tries to give a concise description of the WAM. Appendix B describes the SICStus instruction set and techniques used in it. Appendix C describes the opcodes used in the implemented abstract machines.

2 Prolog

2.1 The language

Prolog (from PROgramming in LOGic) is a declarative language. Code is expressed in facts, rules and questions and the order of statements is often irrelevant. Prolog is in this matter quite different from imperative languages.

Prolog was created in the 1970's and has developed from being used solely as a theorem prover to a complete programming language. A good book about Prolog programming is [5].

2.2 History of SICStus Prolog

The first Prolog interpreter was developed at the University of Marseilles in 1974. The first and second compiler (1977, 1980) were both created in Edinburgh by David H.D. Warren.

The Prolog compilers (interpreters) maintained and developed by SICS are SICStus Prolog and Quintus Prolog. This Master Thesis mainly treats SICStus Prolog (with several modified abstract machines), but experience and conclusions from the implementation of Quintus Prolog have been used as guidelines for how to improve SICStus. All work on SICStus Prolog is currently coordinated by the members of the Intelligent Systems Laboratory (ISL) at SICS[1] in Uppsala.

The version of the code used was the not yet released version 4.0, using the same instruction set as SICStus 3.8. At the time of writing the latest released version of SICStus is version 3.8[2].

[1] The Uppsala group conducts research on finite domain constraint programming and Prolog technology. The group and their work can be found on http://www.sics.se/isl/cps/

[2] Information on how to obtain SICStus as well as well as information on which currently is the latest available release can be found at http://www.sics.se/sicstus/

SICStus code is written in Prolog and C.

3 WAM

3.1 "The" Abstract machine for Prolog

In 1983 David H. D. Warren wrote a technical report [16] on an abstract machine for execution of Prolog programs. The description was not aimed at a broader audience since Warren did not believe that it would be of great interest. Contrary to his beliefs, the abstract machine found its way into many implementations of Prolog such as SICStus, Quintus, XSB, dProlog and Yap and has become the *de facto* implementation vehicle for emulated Prolog systems. The increased interest in the machine and the style of Warren's original text led Hassan Aït-Kaci to do a tutorial reconstruction [2] of his work in 1991. His tutorial recreates the original machine in steps, giving explanations for the design decisions, but it lacks some of the historical/chronological motivations of Warren's paper.

A concise description of the WAM is given in an article by P. Weemeeuw and B. Demoen [17].

3.2 WAM instructions

WAM is an abstract (or virtual) machine, which is register-based. In implementations WAM code acts as an intermediate language between compilation and emulation. Code is first compiled to virtual machine code and then emulated.

The virtual instructions can be classified into a few groups. Hassan Aït-Kaci's tutorial reconstruction [2] of the WAM divides the machine instructions into groups according to their usage.

- **Put** instructions; variable, value, structure, list, constant and unsafe_value.
- **Get** instructions; variable, value, structure, list and constant.
- **Unify** instructions; variable, value, local_value, constant and void.
- **Control** instructions; allocate, deallocate, call, execute and proceed.

These four groups along with the **choice, indexing,** and **cut**[3] instructions, comprise the basic WAM instructions. The choice instructions are used for backtracking, the cut instruction explicitly prevents all backtracking beyond a certain execution point.

Indexing is a technique for optimizing clause selection. Many predicates can be discriminated by their first argument, because of the way code is written. This implies that unification of predicates with more than one clause in the definition can benefit from searches for matches using the first argument as an index.

The outline of the machine together with a more detailed description of the instructions are available in Appendix A.

[3]The cut-instructions were not a part of Warren's original machine.

3.3 SICStus Prolog specifics

WAM's instruction set [16] is extended in SICStus to obtain better performance. Appendix B describes the instructions. The main modifications to the WAM, done in SICStus Prolog, are instruction merging (combinations).

Specializations of merged instructions have also been done. By combining instructions it has also been possible to make instructions obsolete by implementing all its possible combinations. The instructions `allocate` and `deallocate` have by these means been removed in SICStus. This is possible since instructions have been created for `allocate` and `deallocate`, combined with all instructions that can possibly follow in the code, creating one merged instruction for each pair. The result is that less instruction dispatches need to be performed and the original, now obsolete, `allocate` and `deallocate` instructions can be removed from the instruction set.

4 Emulators and their techniques

4.1 Emulators and virtual machines

Compilers can be constructed in different ways. One common solution is to let the compiler compile the code to native code, i.e., code that is specific for the machines architecture, or the assembly language used on the machine. This native code then runs only on the specific machines it is generated for. The disadvantage is that if the code is to run on different platforms, several back-ends might have to be maintained and supported. The advantage is that this results in fast execution of the compiled program.

To avoid having to generate several versions many Prolog systems use an emulator. Emulators have a virtual machine and code is generated for this non-physical machine. The code is first compiled to byte code of the virtual machine. Emulation of this byte code then performs the mapping to the actual machine code instructions. This implementation is less platform dependent and if the emulator is written in a portable language, the solution is fully portable. The main problem is that it is hard to achieve the same execution speed as with native code compilation.

More about compilation techniques can be found in [1].

4.2 Techniques for virtual machines

4.2.1 Extending the instruction set with Combinations and Specializations

Merging several instructions into one creates combinations. This techniques saves dispatches, since one call is enough for all instructions in the combined opcode. Combinations also save space in the generated code, but generally make the emulator grow, which in turn slows down interpretation. Sometimes all possible cases can be covered by the combinations rendering the original instruction obsolete.

Specialization of an abstract instruction splits it into several opcodes, each dealing with a special case. Usually there is also a need for a general catch-all case. Specializations can save time for example if the destination register is known and not needed as an argument. The downside is more operation codes. Specializations can also be done for particular argument types such as constants or nil-valued arguments.

4.2.2 Profiling and static pattern matching

Profiling can be used to optimize the compilation. Profiles of the most frequent predicates, the predicates where most of the time is spent etc are helpful. It is possible to look at the code the compiler for the virtual machine produces and focus on speeding up the most frequently occurring patterns. In particular one could look at the whole code produced and count frequencies of instructions and instruction pairs. Frequent instructions can be used as candidates for specializations and frequent pairs can be used as candidates for combinations. The same technique could be used for triples, but more practical might be to do multiple runs after introducing an improvement. (It might be easier to, after introducing a few combinations, again collect frequency data and find new candidates. Counting triples and merging three instructions at the time might be ineffective.)

4.2.3 Threading

Since ANSI C does not support threading it might be a relevant test to turn it off before running benchmarks. This would also make improvements count more, especially the ones caused by dispatches, and then be easier to detect. Direct threading is described in [9] and was introduced in 1973 in [4]. In virtual machines direct threaded code is used as in assembly language. Each instruction to be executed either contains the code for fetching the next instruction, or has a pointer to a shared copy of the code for fetching the next instruction.

The threading used in SICStus is a type of indirect threading (token threading). Each instruction dispatch consists of three steps:

1. Load next opcode, 2 bytes.
2. Load program address (function of opcode and R/W mode).
3. Jump to code

Step 1 corresponds to the PREFETCH macro in WAM [16]. Steps 2 and 3 correspond to the JUMP_R and JUMP_W macros [16].

4.2.4 Fetches

The instruction merging and instruction specializations give speed-up due to less instruction fetching and less argument decoding, respectively. The drawback is that a larger set of instructions can result in an increase in "instruction-cache miss-rate" [11]. For some implementations a limit on the number of instructions could be a problem, this is not the case for either SICStus or Quintus Prolog. 189 instructions are used by SICStus and the limit for the number is the trade-off fetching/cache-miss.

Hardware specifics (cache and memory sizes) also give rise to bottlenecks when the instruction set becomes larger than the size of the stack frame. The threading technique uses a jump table and extra overhead can be introduced generating a large penalty for fetching of local variables, (if the stack frame becomes too large and the jump-table is in the stack frame,) as is the case in SICStus.

4.2.5 Order of performing combinations and specializations

The order in which optimizations are applied is important when it is possible to both combine and specialize a sequence of instructions. A specialization can prevent a combination from happening, and vice versa. In general the combinations have been put first and their use preferred to that of the specializations. If the hypothesis, that instruction fetches are the major time consumer, is correct, this is the best ordering since it minimizes fetches, but if the fetches are not the main factor, then a different approach could be more efficient. Data presented in this thesis support the fact that instruction fetching is a major time consumer.

4.2.6 Combinations created to match functionality

Certain functionality can be improved by "hand emulating" code with the functionality. By inspecting the resulting code one can find certain combination and specializations that would perform maybe the whole task in one or two abstract machine instructions. If the functionality is highly used in the programs this can give good performance improvement. However, using this technique one need to be careful not to make the machine too program specific.

4.2.7 Simplification gains

Some things work better on simpler abstract machines. In general it is the overhead that is reduced. These improvements are usually small compared to previously described optimizations, as long as no hardware or software thresholds are surpassed. I.e., a machine using less flags require less time, since it does not need to test for their value.

A simpler machine can give some overhead gain by allowing less tests. Tests that are needed for larger instruction sets can be removed if they are no longer used in the de-optimized machine. Such a test could be checking the length of an operand.

4.3 Other optimizations

Some optimizations can be applied at compile time. Such an optimization is postponing `allocate` until as late as possible. Savings are done by executing instructions that can lead to backtracking first, avoiding wasteful `allocates`.

The fact that some instructions bind the value of a register to itself or move the value of one register onto itself is also exploited in many compilers. Instructions that perform such an unnecessary action can be omitted. This technique can be used extensively to reduce the number of moves required. It might also be desirable to minimize the number of registers used. It could also be beneficial to generate code that is amenable to instruction merging. Inline compilation is another technique used extensively in compilers.

Such improvements are done in SICStus Prolog, but most of them are beyond the scope of this thesis.

5 Benchmarks

For any emulator-based implementation there are certain things one needs to focus on. The three most important are (from the point of view that this thesis takes); *emulator size*, *runtime* of the benchmark suite and *size* of the emulated code.

5.1 Code efficiency

To accurately represent CPU execution time, size of code and instruction counts, an appropriate benchmark suite has to be used. The benchmark suite can be used for evaluating the impact of changes in the Virtual Machine.

A problem with benchmarks is that many of those most commonly used are quite small and do not always represent the behavior of "real world" programs. The time measurements also become less accurate for small benchmarks, since caching effects have a greater, or at least a more uneven impact. Despite the disadvantages of small benchmarks, they are used in many cases ([14], [6], [11] and [7]) either in part or completely, so it was decided to use a suite of well known small benchmarks, together with some large benchmarks in this report, to make comparisons possible between this work and future work as well as previous work. The low availability of large benchmarks with good properties is another reason for using small easily available ones. Small benchmarks usually test a certain feature and that makes it easier to trace results, due to changes, to their source. They do not, however, show how well improvements scale, and that is why large benchmarks are needed. Most tests of this kind ([14], [6]) have used, at least, the small benchmark set Aquarius ([15]) suggested by Van Roy. The set used in this research also contains some bigger benchmarks, namely, the SICStus compiler itself, BAM (Berkeley Abstract Machine) as well as certain Finite State Automata tests by Gertjan van Noord, see *http://odur.let.rug.nl/˜vannoord/fsa/fsa.html*.

The following benchmarks were used:

1. **Aquarius** suite: A benchmark suite consisting of many small well known programs; boyer, browse, chat-parser, crypt, deriv, divide10, fast_mu, flatten, log10, meta_qsort, mu, nand, nreverse, ops8, poly, prover, qsort, queens_8, query, reducer, sdda, sendmore, serialise, simple_analyzer, tak, times10, unify and zebra. The number of runs of each benchmark were weighted to give approximately the same execution time. See [15] for reference to the Aquarius suite.

2. **SICStus Prolog:** This benchmark consists in compiling the SICStus 3.8 Prolog compiler. The benchmark actually measure the penalty for increased complexity of the abstract machine, since the expanded abstract machines generally make the compilation slower. For SICStus user manual, see [8].

3. **FSA** (Finite State Automata) utilities: A collection of tools for manipulating finite-state automata, regular expressions and finite-state transducers. The standard FSA tests used were test1 and test3. More information on these utilities and the sources to the benchmarks can be found at: *http://odur.let.rug.nl/˜vannoord/fsa/fsa.html*.

4. **BAM** Berkeley Abstract Machine: Compilation of the Berkeley Abstract Machine and a somewhat I/O related test-run on it. Because of reads and writes to files, time measurements partly depend on the speed of I/O, which is not what this project seek to investigate. The benchmark was kept in the suite to provide the most broad and close to real life spectrum of the suite as possible. Reference available, see [15].

5. **XSB** WAM based Prolog compiler: The benchmark is a compilation of the XSB compiler by itself. For the XSB manual see reference [13].

5.2 Emulator size

The UNIX shell command `nm` offers a way to measure the emulator size (size of the executable) in a more accurate way than simply looking at the size of the object file. This shows the size differences between the machines in a clearer way.

6 Methodology

6.1 Goals

The intention was to develop a methodology for finding candidates for worthwhile optimizations and a method for semi-automatic implementation and optimization of them. The goals were partly achieved although more time would be required to achieve more effective and automated ways of finding candidates.

6.2 Methods

Certain known methods were used, such as counting dynamic instruction pairs appearing in the code, counting frequency of each instruction and optimizing certain functionality. There was no real new method invented, rather used methods were further developed and used together. The focus turned to evaluation of whether specialization or combination of instructions could be the most fruitful.

6.3 Implementation

To test improvements in a quantitative way, a spectrum of the optimizations were implemented and the result of running the benchmarks on them compared to see which improvement yielded the best result. In part this corresponded to finding superoperators [12], but also to try and determine whether specializations or combinations achieved the best improvements. Four different versions of the abstract machine used in SICStus were implemented and evaluated. The implementation process was found to be a lengthier process than expected. Combinations were found to be harder than specialization but the sheer number of specializations made them take longer time to implement. Once implemented though, specialization demanded very little debugging.

6.4 Execution and scripts

Several versions of the code had to be used, one code-tree for each abstract machine. For each code-tree several compilations were necessary. Time measurements and time independent (and time consuming) variables such as space usage and instruction counts are conflicting. To enable memory measurements a compilation flag had to be set, but such versions impose overhead and cannot be used for accurate execution-time measurements. So for each type of test a specific version had to be used. The two platforms used also required separate versions, compiled on the specific platform. The amount of versions needed made the testing more cumbersome, but hopefully also more accurate.

Scripts were used to run the benchmarks and generate the performance data reported. This is advisable since it streamlines testing. Some of the tests show high variance in execution time from one run to another. Reordering and restarting the Prolog version, between each test, helped to get more stable results. A technique for getting

the results more consistent would be to run N runs and pick the one with the lowest, or second lowest result. (In [14] the best of seven tests was picked.) Also reordering the benchmarks between each run would help. Due to lack of time the results from only one run are presented in this thesis. Extraordinary runs are excluded, though.

7 Machines considered

7.1 Abstract machines (Appendix C contains more thorough descriptions)

7.1.1 M_0 - "Warren Abstract Machine"

The first machine considered was the "de-optimized" SICStus, M_0. By removing most prior optimizations, an almost bare WAM was uncovered. `Allocate` and `deallocate` were reintroduced.

The WAM contains 35 instructions. The implementation also uses many extra operation codes to support floats and long integers. It uses special instructions to deal with binding unbound variables to allow for garbage collection. Operation codes used that invoke new variables initializes these.

There are also alignment issues, which means that many operation codes have to exist in two versions. Implementation of the cut instruction and some other technicalities also introduce operation codes. M_0 consists of 136 operation codes and is the starting point for improvements to the SICStus machine.

Indexing is not done in a separate instruction, but rather an incorporated feature of many instructions. Appendix B explains how the SICStus abstract machine works.

7.1.2 M_1 - SICStus 3.8 Abstract Machine

The SICStus 3.8 instruction set was next to be investigated. SICStus virtual machine contains 189 operation codes (opcodes). It has extended the Warren Abstract Machine with optimizations such as several instructions combined to one and instructions specialized for certain frequently occurring cases. In some cases (namely `allocate` and `deallocate`) the combinations/specializations cover all cases and the original WAM instruction can be removed. Indexing is handled as in M_0.

7.1.3 M_2 - Quintus Abstract Machine

Quintus is SICS's other Prolog System. The emulator has a large instruction set (approximately ten times that of SICStus). The M_2 machine was built on ideas from Quintus and contains 427 opcodes. Most improvements are in the form of specialized instructions. As in Quintus Prolog, instructions are specialized for the four first registers, because on some architectures these are directly mapped to hardware registers.

7.1.4 M_3 - Specialized Abstract Machine

An optimization of M_1 built on specializations. The specializations picked are the most frequent instructions from table 9 that easily could be specialized. The other optimization performed was to have all `put_x_value` opcodes translated into a `get_x_variable` opcode with the arguments reversed. In this way the obsolete `put_x_value` could be

removed. The 189 opcodes used in M_1 without `put_x_value`, but with 5 specialized opcodes, results in a total of 244 opcodes.

7.1.5 Optimized Abstract Machine

To come up with an optimized machine was one of the goals of this work, to improve SICStus virtual machine. The machine was to be a combination of the best from SICStus and Quintus Prolog with added improvements deduced from collected data, added specializations and combinations.In short a machine built as a result of the data obtained from the previous machines M_0, M_1, M_2 and M_3.

Unfortunately this improved SICStus was not completely implemented due to lack of time. Instead some recommendations on how this can be done is given in Chapter 9.3.

7.2 Hardware and software

7.2.1 The platforms

Two platforms were used for the tests.

1. SUN Ultra SPARC multiprocessor (8 processors) at 248 MHz, running Solaris 2.7. Referred to as the Sparc architecture in this text.

2. i686 dual processor at 600 MHz, running Red Hat Linux release 6.1 (Cartman) Kernel 2.2.13. Referred to as the x86 architecture in this text.

7.2.2 Registers

All program variables and WAM registers can usually not be mapped directly to hardware registers (because usually there are not enough of them), but it is highly recommended that at least the Program Counter (PC in [2] and [16] called P) is mapped directly to hardware registers. It is often done automatically by the compiler, such as gcc. On some architectures with few hardware registers, like the x86 architecture, a manual register allocation might be needed. In XSB and dProlog the BX register is mapped to PC and in Yap the BP register is used as PC. SICStus forces less important information into memory, thus usually keeping a register free for PC.

8 Performance

Data was collected for CPU time used to run each benchmark and bytecode size of benchmarks. Counts of dynamic frequency of instructions, as well as rate of dynamic occurrence of pairs of instructions were also collected for the benchmarks. The size of the emulator was also measured.

Shell scripts were used to run the tests and get the statistics for each variant of the WAM.

The main problems comparing the results are believed to be due to caching effects. This only applies to the time measurements, code size can be measured accurately. The CPU time measurements should also have been deterministic, but they varied, most probably due to cache effects since paging time is accounted for by the tests themselves.

There was also a problem with the first benchmark run in the suite. It is thought that the machine load can affect the time measurements. Some of the benchmarks vary 100% in one run compared to another. Contention for cache and primary memory could be the factors that create the very uneven figures, especially for the first benchmark in a long series of benchmarks.

8.1 Execution time

The tables present execution times in milli-seconds (both the total and for each benchmark separately) for each virtual machine considered. The absolute values are shown and in parentheses are the relative values compared to M_0. Relative values are obtained by dividing the absolute value of the machine with the corresponding M_0 absolute value and are given with three significant figures.

Execution times are given for the benchmark suite, both for the Sparc architecture and the x86 architecture and also both with and without threaded code. The tables present the data for each machine that is for each version of SICStus virtual instruction set.

Very small benchmarks have been marked by an asterisk. The fastest machine for the sum of the Aquarius suite, all the large benchmarks and the total have been marked by w, for winner.

Table 1, 2 and 4 show that SICStus execution times are almost 10% better than an almost bare WAM.

8.1.1 Threaded

Tables 1 and 2 show the execution times measured using threaded code. Table 1 shows the results on the Sparc machine, and Table 2 the results on the x86 machine.

One source of speed up is fewer dispatches for the merged instructions, especially in benchmarks executing a lot of simple operations a lot of time is wasted on instruction dispatches. Decreased total execution time when introducing combinations shows this.

8.1.2 Not threaded

The benchmarks were also conducted with threading turned off, see tables 3 and 4. As instruction fetches take more time without threading it was expected that combinations would give better speed up and that the effect of specializations would diminish. A machine with many combinations would have made the evaluation easier, but it seems clear that machines with many specializations lose more. It is definitely clear that specializations do not pay off when threading is turned off.

The bad performance of M_2 in Table 4 is hard to explain. The non threaded versions will be less local, all instructions handing control to a big switch statement. This could be something that penalizes a large emulator like M_2. Pipelining and other prediction methods might also work less well, particularly on the x86 architecture.

8.2 Space usage

In Table 5 the impact on the byte-code size is shown. The size difference is due to the more compact code generated by merged and specialized instructions. The table gives the compiled byte-code size in bytes for each virtual machine considered, both the total and for each benchmark separately. The absolute values are given and in parentheses

Sparc execution time in msec for each instruction set					
Benchmark	**Iterations**	M_0	M_1	M_2	M_3
boyer	10	4920(1.00)	*4780(.972)*	*5210(1.06)*	*4920(1.00)*
browse	*5*	*3490(1.00)*	*3290(.943)*	*3410(.977)*	*3280(.940)*
chat_parser	*40*	*4550(1.00)*	*4910(1.08)*	*4890(1.07)*	*4310(.947)*
crypt	*1200*	*3680(1.00)*	*3560(.967)*	*4090(1.11)*	*3600(.978)*
deriv	*50000*	*4940(1.00)*	*4510(.913)*	*4430(.897)*	*4200(.850)*
divide10 (*)	*50000*	*2850(1.00)*	*2590(.909)*	*2900(1.02)*	*2430(.853)*
fast_mu	*5000*	*4350(1.00)*	*4370(1.005)*	*4960(1.14)*	*4400(1.01)*
flatten	*8000*	*4590(1.00)*	*4550(.991)*	*4780(1.04)*	*4730(1.03)*
log10 (*)	*100000*	*3040(1.00)*	*2940(.967)*	*3180(1.05)*	*2840(.934)*
meta_qsort	*1000*	*4200(1.00)*	*4370(1.04)*	*7630(1.82)*	*4360(1.04)*
mu	*6000*	*4340(1.00)*	*4060(.935)*	*4480(1.03)*	*3980(.917)*
nand	*250*	*4760(1.00)*	*4460(.937)*	*4910(1.03)*	*4540(.954)*
nreverse	*15000*	*4740(1.00)*	*3550(.749)*	*3990(.842)*	*3520(.743)*
ops8 (*)	*100000*	*4390(1.00)*	*3810(.868)*	*4400(1.00)*	*3650(.831)*
poly_10 (*)	*100*	*3900(1.00)*	*3440(.882)*	*3450(.885)*	*3670(.941)*
prover	*5000*	*4300(1.00)*	*4020(.935)*	*4480(1.04)*	*3870(.900)*
qsort	*8000*	*4380(1.00)*	*4020(.918)*	*4200(.959)*	*3570(.815)*
queens_8	*100*	*4750(1.00)*	*4240(.893)*	*4440(.935)*	*4130(.869)*
query	*1500*	*4510(1.00)*	*4650(1.03)*	*4700(1.04)*	*4480(.993)*
reducer	*200*	*5630(1.00)*	*5170(.918)*	*5300(.941)*	*4920(.874)*
sdda	*13000*	*4090(1.00)*	*4140(1.01)*	*4580(1.12)*	*4180(1.02)*
sendmore	*60*	*4330(1.00)*	*4020(.928)*	*4350(1.00)*	*3950(.912)*
serialise	*14000*	*5310(1.00)*	*4890(.921)*	*5270(.992)*	*4290(.808)*
simple_analyser	*250*	*4200(1.00)*	*4010(.955)*	*4260(1.01)*	*4050(.964)*
tak	*40*	*4520(1.00)*	*3990(.883)*	*4460(.987)*	*4020(.889)*
times10 (*)	*100000*	*5680(1.00)*	*4520(.796)*	*5340(.940)*	*4360(.768)*
unify	*2500*	*4450(1.00)*	*3890(.874)*	*4620(1.04)*	*3980(.894)*
zebra	*150*	*4350(1.00)*	*4050(.931)*	*4220(.970)*	*4460(1.03)*
Aquarius total		**123240(1.00)**	**114800(.932)**	**126930(1.03)**	**112690(.914)w**
SICStus	*1*	*5700(1.00)*	*5590(.981)w*	*6000(1.05)*	*5840(1.025)*
FSA I	*1*	*25560(1.00)*	*22920(.897)w*	*24270(.950)*	*23600(.923)*
FSA III	*1*	*367270(1.00)*	*332100(.904)w*	*358110(.975)*	*343100(.934)*
BAM	*1*	*131820(1.00)*	*127880(.970)w*	*136800(1.04)*	130180(.988)
XSB	*1*	*10600(1.00)*	*10260(.968)w*	*10940(1.03)*	10540(.994)
Total suite (except Aquarius)		**540950(1.00)**	**498750(.922)w**	**536120(.991)**	**513260(.945)**

Table 1: Execution times in milliseconds, on the Sparc machine, for the different machines. Both absolute values in milliseconds and values relative to M_0 are given. The overall winner is M_1 on the Sparc machine.

x86 execution time in msec for each instruction set					
Benchmark	**Iterations**	M_0	M_1	M_2	M_3
boyer	10	*2150(1.00)*	*1810(.842)*	*1700(.791)*	*1910(.888)*
browse	*5*	*1370(1.00)*	*1200(.876)*	*1080(.788)*	*1100(.803)*
chat_parser	*40*	*2120(1.00)*	*2090(.986)*	*2230(1.05)*	*2140(1.01)*
crypt	*1200*	*1460(1.00)*	*1470(1.01)*	*1360(.932)*	*1440(.986)*
deriv	*50000*	*1860(1.00)*	*1680(.903)*	*1570(.844)*	*1660(.892)*
divide10 (*)	*50000*	*1090(1.00)*	*990(.908)*	*950(.872)*	*1020(.936)*
fast_mu	*5000*	*1970(1.00)*	*1890(.959)*	*2110(1.07)*	*1880(.954)*
flatten	*8000*	*2100(1.00)*	*2010(.957)*	*2150(1.02)*	*2110(1.00)*
log10 (*)	*100000*	*1130(1.00)*	*1150(1.02)*	*1080(.956)*	*1140(1.01)*
meta_qsort	*1000*	*1740(1.00)*	*1660(.954)*	*1520(.874)*	*1670(.960)*
mu	*6000*	*1650(1.00)*	*1330(.806)*	*1360(.824)*	*1320(.800)*
nand	*250*	*2070(1.00)*	*1980(.957)*	*2030(.981)*	*1950(.942)*
nreverse	*15000*	*1540(1.00)*	*1150(.747)*	*970(.630)*	*1000(.649)*
ops8 (*)	*100000*	*1760(1.00)*	*1640(.932)*	*1550(.881)*	*1620(.920)*
poly_10 (*)	*100*	*1630(1.00)*	*1380(.847)*	*1200(.736)*	*1340(.822)*
prover	*5000*	*1890(1.00)*	*1790(.947)*	*1760(.931)*	*1720(.910)*
qsort	*8000*	*1600(1.00)*	*1250(.781)*	*1380(.862)*	*1290(.806)*
queens_8	*100*	*1710(1.00)*	*1610(.942)*	*1460(.854)*	*1710(1.00)*
query	*1500*	*1830(1.00)*	*1730(.945)*	*1660(.907)*	*1830(1.00)*
reducer	*200*	*2370(1.00)*	*2220(.937)*	*2160(.911)*	*2270(.958)*
sdda	*13000*	*1930(1.00)*	*1960(1.02)*	*2250(1.17)*	*2050(1.06)*
sendmore	*60*	*1770(1.00)*	*1500(.847)*	*1400(.791)*	*1450(.819)*
serialise	*14000*	*1990(1.00)*	*1730(.869)*	*1800(.905)*	*1830(.920)*
simple_analyser	*250*	*1970(1.00)*	*1920(.975)*	*2120(1.08)*	*2020(1.025)*
tak	*40*	*1730(1.00)*	*1710(.988)*	*1500(.867)*	*1680(.971)*
times10 (*)	*100000*	*2040(1.00)*	*1810(.887)*	*1680(.824)*	*1710(.838)*
unify	*2500*	*1760(1.00)*	*1640(.932)*	*1630(.926)*	*1670(.949)*
zebra	*150*	*1860(1.00)*	*1890(1.02)*	*1860(1.00)*	*1870(1.01)*
Aquarius total		**50090(1.00)**	**46190(.922)**	**45520(.909)w**	**46400(.926)**
SICStus	*1*	*3020(1.00)*	*2840(.940)w*	*3140(1.04)*	*3000(.993)*
FSA I	*1*	*12110(1.00)*	*11270(.931)w*	*11300(.933)*	*11390(.941)*
FSA III	*1*	*163460(1.00)*	*145310(.889)*	*140600(.860)w*	*141260(.864)*
BAM	*1*	*60310(1.00)*	*60300(1.00)*	*65370(1.08)*	*59130(.980)w*
XSB	*1*	*4960(1.00)*	*4590(.925)*	*4690(.946)*	*4580(.923)w*
Total suite (except Aquarius)		**243860(1.00)**	**224310(.920)**	**225100(.923)**	**219360(.900)w**

Table 2: Execution times in milli seconds, on the x86 machine, for the different machines. The overall winner is M_3 on the x86 machine!

Sparc execution time in msec with threading disabled					
Benchmark	**Iterations**	M_0	M_1	M_2	M_3
boyer	*10*	*6970(1.00)*	*5450(.782)*	*5650(.811)*	*5300(.760)*
browse	*5*	*4640(1.00)*	*3890(.838)*	*4130(.890)*	*3830(.825)*
chat_parser	*40*	*5060(1.00)*	*4830(.955)*	*4990(.986)*	*4990(.986)*
crypt	*1200*	*4320(1.00)*	*4420(1.02)*	*3920(.907)*	*3940(.912)*
deriv	*50000*	*5900(1.00)*	*4850(.822)*	*5150(.873)*	*4900(.831)*
divide10 (*)	*50000*	*3640(1.00)*	*2860(.786)*	*3170(.871)*	*2940(.808)*
fast_mu	*5000*	*5370(1.00)*	*5090(.948)*	*5460(.1017)*	*4910(.914)*
flatten	*8000*	*5900(1.00)*	*4880(.827)*	*5490(.931)*	*4950(.839)*
log10 (*)	*100000*	*5310(1.00)*	*3170(.597)*	*3540(.667)*	*3450(.650)*
meta_qsort	*1000*	*5170(1.00)*	*4890(.946)*	*4810(.930)*	*4360(.843)*
mu	*6000*	*4940(1.00)*	*4600(.931)*	*4820(.976)*	*4900(.992)*
nand	*250*	*5770(1.00)*	*5060(.877)*	*5640(.977)*	*4940(.856)*
nreverse	*15000*	*6810(1.00)*	*4320(.634)*	*4950(.727)*	*4720(.693)*
ops8 (*)	*100000*	*5590(1.00)*	*4430(.792)*	*4850(.868)*	*4380(.784)*
poly_10 (*)	*100*	*4640(1.00)*	*4060(.875)*	*4570(.985)*	*3920(.845)*
prover	*5000*	*4950(1.00)*	*4560(.921)*	*4680(.945)*	*4480(.905)*
qsort	*8000*	*5220(1.00)*	*4560(.874)*	*4690(.898)*	*4780(.916)*
queens_8	*100*	*5950(1.00)*	*4920(.827)*	*5160(.867)*	*4890(.822)*
query	*1500*	*6700(1.00)*	*5050(.754)*	*5040(.752)*	*5310(.793)*
reducer	*200*	*6520(1.00)*	*6410(.983)*	*6160(.945)*	*6440(.988)*
sdda	*13000*	*4960(1.00)*	*4680(.944)*	*5020(1.01)*	*4760(.960)*
sendmore	*60*	*4920(1.00)*	*4810(.978)*	*4960(1.01)*	*4590(.933)*
serialise	*14000*	*6120(1.00)*	*5760(.941)*	*5620(.918)*	*5480(.895)*
simple_analyser	*250*	*5050(1.00)*	*4460(.883)*	*5040(.998)*	*4630(.917)*
tak	*40*	*4860(1.00)*	*4550(.936)*	*4710(.969)*	*4850(.998)*
times10 (*)	*100000*	*6670(1.00)*	*5030(.754)*	*5650(.847)*	*5340(.801)*
unify	*2500*	*5160(1.00)*	*4620(.895)*	*4910(.952)*	*4580(.888)*
zebra	*150*	*4750(1.00)*	*4600(.968)*	*4690(.987)*	*4670(.983)*
Aquarius total		**151860(1.00)**	**130810(.861)**	**137470(.905)**	**131230(.864)**
SICStus	*1*	*6790(1.00)*	*6040(.890)*	*6860(1.01)*	*6190(.912)*
FSA I	*1*	*28410(1.00)*	*26250(.924)*	*28060(.988)*	*26940(.948)*
FSA III	*1*	*444380(1.00)*	*395000(.889)*	*459550(1.03)*	*386800(.870)*
BAM	*1*	*160650(1.00)*	*143860(.895)*	*178740(1.11)*	*148240(.923)*
XSB	*1*	*13770(1.00)*	*11370(.826)*	*12850(.933)*	*11430(.830)*
Total suite (except Aquarius)		**654000(1.00)**	**582520(.891)**	**674460(1.03)**	**579600(.886)**

Table 3: Execution times in milli seconds, on the Sparc architecture, for the different machines. Here threading is disabled. The overall winner is M_3 due to the high impact of FSA III.

x86 execution time in msec with threading disabled					
Benchmark	**Iterations**	M_0	M_1	M_2	M_3
boyer	*10*	*2470(1.00)*	*2340(.947)*	*2620(1.06)*	*2350(.951)*
browse	*5*	*1800(1.00)*	*1630(.906)*	*1690(.939)*	*1620(.900)*
chat_parser	*40*	*2330(1.00)*	*2260(.970)*	*2500(1.07)*	*2310(.991)*
crypt	*1200*	*1710(1.00)*	*1680(.982)*	*1600(.936)*	*1680(.982)*
deriv	*50000*	*2310(1.00)*	*2120(.918)*	*2220(.961)*	*2110(.913)*
divide10 (*)	*50000*	*1360(1.00)*	*1230(.904)*	*1360(1.00)*	*1270(.934)*
fast_mu	*5000*	*2280(1.00)*	*2230(.978)*	*2520(1.13)*	*2250(.987)*
flatten	*8000*	*2420(1.00)*	*2440(1.01)*	*2870(1.19)*	*2570(1.06)*
log10 (*)	*100000*	*1420(1.00)*	*1400(.986)*	*1610(1.13)*	*1430(1.01)*
meta_qsort	*1000*	*2220(1.00)*	*1980(.892)*	*2120(.955)*	*2050(.923)*
mu	*6000*	*2150(1.00)*	*2060(.958)*	*2170(1.01)*	*2060(.958)*
nand	*250*	*2430(1.00)*	*2380(.979)*	*2590(1.07)*	*2390(.984)*
nreverse	*15000*	*2480(1.00)*	*2120(.855)*	*2450(.988)*	*2010(.810)*
ops8 (*)	*100000*	*1940(1.00)*	*1860(.959)*	*2050(1.06)*	*1820(.938)*
poly_10 (*)	*100*	*1880(1.00)*	*1610(.856)*	*1860(.989)*	*1620(.862)*
prover	*5000*	*2180(1.00)*	*2070(.950)*	*2300(1.06)*	*2050(.940)*
qsort	*8000*	*2160(1.00)*	*1980(.917)*	*2160(1.00)*	*1990(.921)*
queens_8	*100*	*2180(1.00)*	*2130(.977)*	*2010(.922)*	*2150(.986)*
query	*1500*	*2120(1.00)*	*1970(.929)*	*1990(.939)*	*1940(.915)*
reducer	*200*	*2920(1.00)*	*2920(1.00)*	*3000(1.03)*	*2870(.983)*
sdda	*13000*	*2220(1.00)*	*2250(1.01)*	*2530(1.14)*	*2370(1.07)*
sendmore	*60*	*2120(1.00)*	*2140(1.01)*	*2100(.991)*	*2060(.972)*
serialise	*14000*	*2720(1.00)*	*2440(.897)*	*2550(.938)*	*2410(.886)*
simple_analyser	*250*	*2260(1.00)*	*2280(1.01)*	*2740(1.21)*	*2400(1.06)*
tak	*40*	*2090(1.00)*	*2110(1.01)*	*2180(1.04)*	*2240(1.07)*
times10 (*)	*100000*	*2430(1.00)*	*2270(.934)*	*2520(1.04)*	*2250(.926)*
unify	*2500*	*2100(1.00)*	*2100(1.00)*	*2400(1.14)*	*2110(1.005)*
zebra	*150*	*2300(1.00)*	*2160(.939)*	*2150(.935)*	*2090(.909)*
Aquarius total		**61000(1.00)**	**58160(.953)w**	**62860(1.03)**	**58470(.959)**
SICStus	*1*	*3270(1.00)*	*3230(.988)w*	*3990(1.22)*	*3290(1.01)*
FSA I	*1*	*13030(1.00)w*	*13150(1.01)*	*14150(1.09)*	*13650(1.05)*
FSA III	*1*	*192720(1.00)*	*181410(.941)w*	*200340(1.04)*	*185090(.960)*
BAM	*1*	*66390(1.00)w*	*69900(1.05)*	*82530(1.24)*	*69480(1.05)*
XSB	*1*	*5770(1.00)*	*5650(.979)w*	*6180(1.07)*	*5570(.965)*
Total suite (except Aquarius)		**281180(1.00)**	**273340(.972)w**	**307190(1.09)**	**277080(.985)**

Table 4: Execution times in milli seconds, on the x86 architecture, for the different machines. Here threading is disabled. The overall winner is M_1 on the x86 machine.

Code size in bytes for each instruction set				
Benchmark	M_0	M_1	M_2	M_3
boyer	*12792(1.00)*	*11976(.936)*	*10928(.854)*	*11320(.885)*
browse	*3384(1.00)*	*3096(.915)*	*2808(.830)*	*3048(.901)*
chat_parser	*38992(1.00)*	*35392(.908)*	*33608(.862)*	*35056(.899)*
crypt	*2056(1.00)*	*1944(.946)*	*1848(.899)*	*1912(.930)*
deriv	*1624(1.00)*	*1520(.936)*	*1344(.828)*	*1472(.906)*
divide10	*1288(1.00)*	*1184(.919)*	*1024(.795)*	*1136(.882)*
fast_mu	*1744(1.00)*	*1624(.931)*	*1576(.904)*	*1608(.922)*
flatten	*4832(1.00)*	*4352(.901)*	*3872(.801)*	*4208(.871)*
log10	*1192(1.00)*	*1088(.913)*	*960(.805)*	*1040(.872)*
meta_qsort	*2312(1.00)*	*2160(.934)*	*2024(.875)*	*2096(.907)*
mu	*1040(1.00)*	*1016(.977)*	*888(.854)*	*960(.923)*
nand	*21792(1.00)*	*19680(.903)*	*18904(.867)*	*19368(.889)*
nreverse	*512(1.00)*	*504(.984)*	*472(.922)*	*496(.969)*
ops8	*1248(1.00)*	*1144(.917)*	*1000(.801)*	*1096(.878)*
poly_10	*2968(1.00)*	*2736(.922)*	*2408(.811)*	*2632(.887)*
prover	*3464(1.00)*	*3112(.898)*	*2936(.848)*	*3064(.885)*
qsort	*792(1.00)*	*768(.970)*	*728(.919)*	*752(.949)*
queens_8	*752(1.00)*	*696(.926)*	*640(.851)*	*680(.904)*
query	*2464(1.00)*	*2448(.994)*	*2384(.968)*	*2448(.994)*
reducer	*10296(1.00)*	*9512(.924)*	*8552(.831)*	*9176(.891)*
sdda	*7512(1.00)*	*6904(.919)*	*6160(.820)*	*6712(.894)*
sendmore	*1928(1.00)*	*1768(.917)*	*1656(.859)*	*1768(.917)*
serialise	*1240(1.00)*	*1128(.910)*	*976(.787)*	*1096(.884)*
simple_analyser	*13912(1.00)*	*12720(.914)*	*11720(.842)*	*12504(.899)*
tak	*408(1.00)*	*392(.961)*	*344(.843)*	*384(.941)*
times10	*1288(1.00)*	*1184(.919)*	*1024(.795)*	*1136(.882)*
unify	*8456(1.00)*	*7720(.913)*	*7016(.830)*	*7568(.895)*
zebra	*1432(1.00)*	*1288(.899)*	*1048(.732)*	1232(.860)
Aquarius total	**151720(1.00)**	**139056(.917)**	**128848(.849)**	**135968(.896)**
SICStus	*194488(1.00)*	*175432(.902)*	*156640(.805)*	170608(.877)
FSA	*212896(1.00)*	*200848(.943)*	*191792(.901)*	*198488(.932)*
BAM	*38768(1.00)*	*34744(.896)*	*32464(.837)*	34272(.884)
XSB	*138912(1.00)*	*124408(.896)*	*113960(.820)*	121824(.877)
Complete suite(except Aquarius)	**585064(1.00)**	**535432(.915)**	**494856(.846)**	**525192(.897)**

Table 5: Byte-code size of the benchmarks suite for the different machines. Both absolute values in bytes and values relative to M_0 are given.

are the relative values compared to M_0. Relative values are obtained by dividing the absolute value of the machine with the corresponding M_0 absolute value and are given with three significant figures.

Specializations save space and the savings increase the more specializations one uses, as expected.

8.3 Dynamic instruction counts

The instruction counts can be very useful. They suggest which way to go, which optimizations to do, to achieve the optimal mergers. In Tables 6, 7, 8 and 9 the 30 most frequent instructions and pairs of instructions for each machine are shown. In Table 10 the instruction frequencies of all machines are compared. The shown sums are for the whole suite of benchmarks.

Certain frequently occurring pairs of instructions belong to different clauses, and cannot therefore be directly considered for mergers. To avoid merging pairs from different clauses when backtracking occurs the `fail` instruction is invoked. This results in the last instruction before the backtracking occurred being paired with `fail` and results in a false or constructed pair. This pair cannot be considered for merger, but on the other hand this number gives a size estimate of how often backtracking occurs. The actual `fail` instruction, when occurring, is included in the same counts. Pairs marked with a †in Table 6, 7, 8 and 9 are either inter-procedural ones or pairs with the `fail` construction. The same `fail` construction also occurs in Table 10; it is kept there to show how often backtracking occurs.

The information from all tables was used extensively during the search for good specializations etc. It was considered good to obtain lower counts for the instruction counts, since that implies less dispatches. The pairs were used to find good candidates for mergers. In M_2, optimizations empirically deduced and used by Quintus Prolog have been used as a model for implementation on top of M_0.

9 Analysis of the results and future work

9.1 Comparing the machines

9.1.1 Time

The results from the Sparc architecture in Table 1 shows that speed is increased by approximately 11%, comparing M_0 to M_1. M_2 gains little, no more than 1%, on the large benchmarks, and actually loses somewhat on the really small ones, compared to M_0. M_1 is clearly the machine that wins the time race on the Sparc architecture. Since time is held in high regard M_1 was selected to be the foundation of the specialized machine, M_3.

The results from the machine M_3 with its specializations, on the Sparc machine Table 1, are relatively disappointing. The specialized machine only shows a slight improvement in bytecode size (2% smaller) but a slower execution time by 2%. Especially disappointing is that a small speedup is noticed in the smallest benchmarks, but the larger the benchmarks get the lower speedups are measured, compared to the original SICStus.

M_0					
Instruction count	Frequency	Instruction pairs		Remark	Frequency
UNIFY_X_VARIABLE	936(18.0%)	UNIFY_X_VARIABLE	UNIFY_X_VARIABLE		379(7.3%)
PUT_X_VALUE	581(11.2%)	PUT_X_VALUE	PUT_X_VALUE		293(5.6%)
EXECUTE	503(9.7%)	PUT_X_VALUE	EXECUTE		242(4.7%)
PUT_Y_VALUE	351(6.7%)	GET_X_VARIABLE	UNIFY_X_VARIABLE		189(3.6%)
GET_X_VARIABLE	259(5.0%)	PUT_Y_VALUE	PUT_Y_VALUE		178(3.4%)
GET_Y_VARIABLE	248(4.8%)	GET_Y_VARIABLE	GET_Y_VARIABLE		171(3.3%)
HEAPMARGIN_CALL	233(4.5%)	UNIFY_X_VARIABLE	HEAPMARGIN_CALL		168(3.2%)
FUNCTION_2	206(4.0%)	UNIFY_X_VARIABLE	PUT_X_VALUE		168(3.2%)
GET_LIST	189(3.6%)	HEAPMARGIN_CALL	FUNCTION_2		160(3.1%)
UNIFY_X_LOCAL_VALUE	167(3.2%)	EXECUTE	GET_X_VARIABLE	†	145(2.8%)
UNIFY_X_VALUE	151(2.9%)	GET_LIST	UNIFY_X_LOCAL_VALUE		133(2.6%)
GET_STRUCTURE	150(2.9%)	EXECUTE	GET_LIST	†	125(2.4%)
PROCEED	90(1.7%)	FUNCTION_2	EXECUTE		119(2.3%)
ALLOCATE	87(1.7%)	UNIFY_X_LOCAL_VALUE	UNIFY_X_VARIABLE		107(2.0%)
FIRSTCALL	87(1.7%)	UNIFY_X_VARIABLE	GET_STRUCTURE		90(1.7%)
DEALLOCATE	78(1.5%)	DEALLOCATE	EXECUTE		78(1.5%)
FUNCTION_2_IMM	70(1.3%)	EXECUTE	UNIFY_X_VARIABLE	†	67(1.3%)
GET_X_VALUE	70(1.3%)	PROCEED	PUT_Y_VALUE	†	65(1.2%)
TRY	63(1.2%)	PUT_Y_VALUE	DEALLOCATE		56(1.1%)
PUT_STRUCTURE	57(1.1%)	GET_STRUCTURE	UNIFY_X_VARIABLE		55(1.1%)
PUT_Y_UNSAFE_VALUE	56(1.1%)	PUT_STRUCTURE	UNIFY_X_VALUE		49(0.9%)
CUTB	51(1.0%)	UNIFY_X_VALUE	UNIFY_X_VALUE		49(0.9%)
FAIL (opcode+inter.proc.calls)	50(1.0%)	ALLOCATE	GET_Y_VARIABLE		44(0.8%)
UNIFY_Y_VARIABLE	46(0.9%)	PUT_Y_UNSAFE_VALUE	PUT_Y_VALUE		43(0.8%)
PUT_CONSTANT	45(0.9%)	UNIFY_X_VARIABLE	GET_LIST		42(0.8%)
PUT_Y_VARIABLE	41(0.8%)	EXECUTE	TRY	†	37(0.7%)
UNIFY_VOID	41(0.8%)	PUT_Y_VALUE	PUT_Y_UNSAFE_VALUE		36(0.7%)
BUILTIN_2	35(0.7%)	GET_X_VARIABLE	PUT_X_VALUE		32(0.6%)
BUILTIN_2_IMM	34(0.6%)	GET_X_VARIABLE	GET_X_VARIABLE		31(0.6%)
BUILTIN_1	30(0.6%)	GET_STRUCTURE	GET_X_VARIABLE		31(0.6%)

Table 6: Instruction frequencies and instruction-pair frequency for the 30 most frequently occurring counts of M_0. Values are given both as absolute counts in **millions** and in percentage of total number of pairs. The shown pairs constitute 65% of M_0 and the shown count is 86%.

M_1						
Instruction count	Frequency	Remark	Instruction pair		Remark	Frequency
EXECUTE	425(11.7%)		GET_X_VARIABLE	U2_XVAR_XVAR	C	187(5.1%)
U2_XVAR_XVAR	297(8.2%)	S	HEAPMARGIN_CALL	FUNCTION_2		160(4.4%)
HEAPMARGIN_CALL	233(6.4%)		U2_XVAR_XVAR	HEAPMARGIN_CALL		154(4.2%)
PUT_XVAL_XVAL	225(6.2%)	S	PUT_XVAL_XVAL	EXECUTE	C	144(4.0%)
GET_X_VARIABLE	212(5.8%)	S	EXECUTE	GET_LIST	†	124(3.4%)
FUNCTION_2	206(5.7%)		FUNCTION_2	EXECUTE		119(3.3%)
GET_LIST	183(5.0%)	S	EXECUTE	GET_X_VARIABLE	†	117(3.2%)
PUT_X_VALUE	131(3.6%)	S	GET_LIST	U2_XLVAL_XVAR	C	106(2.9%)
U2_XLVAL_XVAR	107(2.9%)		PUT_X_VALUE	EXECUTE	C	98(2.7%)
GET_STRUCTURE	100(2.7%)	S	U2_XVAR_XVAR	PUT_XVAL_XVAL		59(1.6%)
FIRSTCALL	87(2.4%)		U2_XLVAL_XVAR	PUT_XVAL_XVAL		58(1.6%)
LASTCALL	78(2.1%)		EXECUTE	U2_XVAR_XVAR	†	57(1.6%)
FUNCTION_2_IMM	70(1.9%)		U2_XLVAL_XVAR	PUT_X_VALUE		48(1.3%)
GET_YVAR_YVAR	66(1.8%)		PUT_XVAL_XVAL	PUT_XVAL_XVAL	C	46(1.3%)
UNIFY_X_VARIABLE	64(1.8%)		PUT_STRUCTURE	U2_XVAL_XVAL		46(1.3%)
TRY	63(1.7%)		GET_STRUCTURE	GET_X_VARIABLE		31(0.8%)
PUT_STRUCTURE	57(1.6%)		FUNCTION_2_IMM	FUNCTION_2_IMM		30(0.8%)
PUT_Y_VALUE	51(1.4%)		GET_YVAR_YVAR	GET_YVAR_YVAR		26(0.7%)
GET_STRUCTURE_XVAR_XVAR	50(1.4%)		U2_XVAR_XVAR	GET_STRUCTURE_XVAR_XVAR		26(0.7%)
FAIL(opcode+inter.proc.calls)	49(1.3%)		GET_STRUCTURE_XVAR_XVAR	GET_STRUCTURE		25(0.7%)
U2_XVAL_XVAL	48(1.3%)		UNIFY_X_VARIABLE	UNIFY_Y_FIRST_VARIABLE		25(0.7%)
PUT_CONSTANT	45(1.2%)		U2_XVAL_XVAL	EXECUTE		23(0.6%)
GET_YFVAR_YVAR	41(1.1%)		GET_LIST	UNIFY_X_VARIABLE		22(0.6%)
GET_X_VALUE	39(1.1%)		PUT_XVAL_XVAL	PUT_X_VALUE	C	22(0.6%)
PROCEED	35(1.0%)		U2_XVAL_XVAL	PUT_STRUCTURE		22(0.6%)
BUILTIN_2	35(1.0%)		FUNCTION_2	PUT_STRUCTURE		22(0.6%)
PUT_Y_VARIABLE	34(0.9%)		PROCEED	LASTCALL	†	22(0.6%)
BUILTIN_2_IMM	34(0.9%)		PUT_Y_VARIABLE	FIRSTCALL		22(0.6%)
GET_Y_VARIABLE	32(0.9%)		PUT_Y_VALUE	HEAPMARGIN_CALL		21(0.6%)
CUTB	32(0.9%)		GET_LIST	U2_XLVAL_XLVAL		21(0.6%)

Table 7: Instruction frequencies and instruction-pair frequency for the 30 most frequently occurring counts of M_1. Values are given both as absolute counts in **millions** and in percentage of total number of pairs. The shown pairs constitute 54% of M_1 and the shown count is 86%.

M_2					
Instruction count	Frequency	Instruction pair		Remark	Frequency
EXECUTE	425(9.2%)	HEAPMARGIN_CALL	FUNCTION_2		160(3.5%)
HEAPMARGIN_CALL	233(5.0%)	FUNCTION_2	EXECUTE		119(2.6%)
FUNCTION_2	206(4.5%)	EXECUTE	GET_LIST	†	108(2.3%)
GET_LIST	129(2.8%)	EXECUTE	GET_AN_VARIABLE_X3	†	102(2.2%)
GET_AN_VARIABLE_X3	128(2.8%)	UNIFY_VARS_X3_XN	HEAPMARGIN_CALL		102(2.2%)
UNIFY_X_VALUE	127(2.7%)	GET_AN_VARIABLE_X3	UNIFY_VARS_X3_XN		102(2.2%)
UNIFY_VARS_X3_XN	113(2.5%)	GET_A0_VARIABLE_X1	GET_A1_VARIABLE_XN		66(1.4%)
U2_XVAR_XVAR	100(2.2%)	GET_LIST	UNIFY_LOCAL_VALUE_X3		65(1.4%)
UNIFY_X_VARIABLE	92(2.0%)	GET_A1_VARIABLE_XN	EXECUTE		62(1.3%)
FIRSTCALL	87(1.9%)	GET_A0_VARIABLE_XN	EXECUTE		61(1.3%)
GET_A1_VARIABLE_XN	85(1.8%)	UNIFY_X_VARIABLE	GET_A0_VARIABLE_X2		60(1.3%)
GET_A0_VARIABLE_XN	84(1.8%)	GET_A3_VARIABLE_X2	UNIFY_VARS_XN_X2		59(1.3%)
GET_A0_VARIABLE_X1	76(1.6%)	UNIFY_VARS_XN_X2	GET_A0_VARIABLE_X1		59(1.3%)
GET_A2_VARIABLE_XN	73(1.6%)	GET_A1_VARIABLE_X3	GET_A2_VARIABLE_XN		51(1.1%)
GET_A0_VARIABLE_X2	71(1.5%)	GET_A3_VARIABLE_XN	EXECUTE		50(1.1%)
FUNCTION_2_IMM	70(1.5%)	UNIFY_VARIABLE_X3	GET_A0_VARIABLE_XN		48(1.0%)
UNIFY_LOCAL_VALUE_X1	70(1.5%)	UNIFY_LOCAL_VALUE_X3	UNIFY_VARIABLE_X3		48(1.0%)
UNIFY_LOCAL_VALUE_X3	69(1.5%)	UNIFY_X_VALUE	UNIFY_X_VALUE		45(1.0%)
PROCEED	68(1.5%)	GET_Y_VARIABLE	GET_Y_VARIABLE		45(1.0%)
TRY	63(1.4%)	GET_A2_VARIABLE_XN	GET_A3_VARIABLE_XN		44(1.0%)
PUT_Y_VALUE	62(1.3%)	GET_A0_VARIABLE_X2	GET_A1_VARIABLE_X3		44(0.9%)
UNIFY_VARIABLE_X3	61(1.3%)	GET_LIST	UNIFY_LOCAL_VALUE_X1		44(0.9%)
GET_A3_VARIABLE_X2	60(1.3%)	UNIFY_LOCAL_VALUE_X1	UNIFY_X_VARIABLE		41(0.9%)
ALLOCATE	60(1.3%)	U2_XVAR_XVAR	HEAPMARGIN_CALL		38(0.8%)
GET_A1_VARIABLE_X3	60(1.3%)	PUT_Y_VALUE	PUT_Y_VALUE		37(0.8%)
UNIFY_VARS_XN_X2	59(1.3%)	FUNCTION_2_IMM	FUNCTION_2_IMM		30(0.6%)
GET_A3_VARIABLE_XN	56(1.2%)	GET_A2_VARIABLE_X3	EXECUTE		29(0.6%)
GET_Y_VARIABLE	54(1.2%)	GET_A2_VARIABLE_XN	EXECUTE		27(0.6%)
GET_STRUCTURE	51(1.1%)	EXECUTE	U2_XVAR_XVAR	†	25(0.5%)
GET_A1_STRUCTURE	51(1.1%)	GET_STRUCTURE	U2_XVAR_XVAR		24(0.5%)
FAIL(opcode+inter.proc.calls)	50(1.1%)				

Table 8: Instruction frequencies and instruction-pair frequency for the 30 most frequently occurring counts of M_2. Values are given both as absolute counts in **millions** and in percentage of total number of pairs. The first 30 shown pairs constitute 39% of M_2 and the shown count is 64%.

M_3					
Instruction count	Frequency	Instruction pair		Remark	Frequency
EXECUTE	425(11.7%)	HEAPMARGIN_CALL	FUNCTION_2		160(4.4%)
GET_XVAR_XVAR	252(6.9%)	GET_XVAR_XVAR	EXECUTE		133(3.7%)
HEAPMARGIN_CALL	233(6.4%)	FUNCTION_2	EXECUTE		119(3.3%)
FUNCTION_2	206(5.7%)	EXECUTE	GET_LIST	†	108(3.0%)
GET_LIST	128(3.5%)	GET_LIST	U2_XLVAL_XVAR		106(2.9%)
GET_AN_VARIABLE_X3	124(3.4%)	EXECUTE	GET_AN_VARIABLE_X3	†	102(2.8%)
UNIFY_VARS_X3_XN	112(3.1%)	UNIFY_VARS_X3_XN	HEAPMARGIN_CALL		102(2.8%)
U2_XLVAL_XVAR	107(2.9%)	GET_AN_VARIABLE_X3	UNIFY_VARS_X3_XN		102(2.8%)
FIRSTCALL	87(2.4%)	GET_XVAR_XVAR	GET_XVAR_XVAR		62(1.7%)
LASTCALL	78(2.1%)	GET_A0_VARIABLE_XN	EXECUTE		61(1.7%)
U2_XVAR_XVAR	74(2.0%)	UNIFY_VARS_XN_X2	GET_XVAR_XVAR		59(1.6%)
GET_A0_VARIABLE_XN	72(2.0%)	GET_A3_VARIABLE_X2	UNIFY_VARS_XN_X2		59(1.6%)
FUNCTION_2_IMM	70(1.9%)	U2_XLVAL_XVAR	GET_XVAR_XVAR		58(1.6%)
GET_YVAR_YVAR	66(1.8%)	U2_XLVAL_XVAR	GET_A0_VARIABLE_XN		48(1.3%)
TRY	63(1.7%)	PUT_STRUCTURE	U2_XVAL_XVAL		46(1.3%)
GET_A3_VARIABLE_X2	59(1.6%)	U2_XVAR_XVAR	HEAPMARGIN_CALL		42(1.2%)
UNIFY_VARS_XN_X2	59(1.6%)	FUNCTION_2_IMM	FUNCTION_2_IMM		30(0.8%)
PUT_STRUCTURE	57(1.6%)	GET_YVAR_YVAR	GET_YVAR_YVAR		26(0.7%)
PUT_Y_VALUE	51(1.4%)	EXECUTE	U2_XVAR_XVAR	†	25(0.7%)
GET_STRUCTURE_XVAR_XVAR	50(1.4%)	U2_XVAL_XVAL	EXECUTE		23(0.6%)
FAIL	49(1.3%)	U2_XVAL_XVAL	PUT_STRUCTURE		22(0.6%)
GET_A1_STRUCTURE	49(1.3%)	U2_XVAR_XVAR	GET_STRUCTURE_XVAR_XVAR		22(0.6%)
U2_XVAL_XVAL	48(1.3%)	GET_STRUCTURE_XVAR_XVAR	GET_A1_STRUCTURE		22(0.6%)
PUT_CONSTANT	45(1.2%)	FUNCTION_2	PUT_STRUCTURE		22(0.6%)
GET_YFVAR_YVAR	41(1.1%)	GET_A1_STRUCTURE	GET_AN_VARIABLE_X3		22(0.6%)
GET_X_VALUE	39(1.1%)	GET_AN_VARIABLE_X3	U2_XVAR_XVAR		22(0.6%)
PROCEED	35(1.0%)	PROCEED	LASTCALL	†	22(0.6%)
BUILTIN_2	35(1.0%)	PUT_Y_VARIABLE	FIRSTCALL		22(0.6%)
PUT_Y_VARIABLE	34(0.9%)	PUT_Y_VALUE	HEAPMARGIN_CALL		21(0.6%)
BUILTIN_2_IMM	34(0.9%)	EXECUTE	TRY	†	21(0.6%)

Table 9: Instruction frequencies and instruction-pair frequency for the 30 most frequently occurring counts of M_3. Values are given both as absolute counts in millions and in percentage of total number of pairs. The shown pairs constitute 47% of M_3 and the shown count is 77%.

M_0	M_1	M_2	M_3
UNIFY_X_VARIABLE	EXECUTE	EXECUTE	EXECUTE
PUT_X_VALUE	U2_XVAR_XVAR	HEAPMARGIN_CALL	GET_XVAR_XVAR
EXECUTE	HEAPMARGIN_CALL	FUNCTION_2	HEAPMARGIN_CALL
PUT_Y_VALUE	PUT_XVAL_XVAL	GET_LIST	FUNCTION_2
GET_X_VARIABLE	GET_X_VARIABLE	GET_AN_VARIABLE_X3	GET_LIST
GET_Y_VARIABLE	FUNCTION_2	UNIFY_X_VALUE	GET_AN_VARIABLE_X3
HEAPMARGIN_CALL	GET_LIST	UNIFY_VARS_X3_XN	UNIFY_VARS_X3_XN
FUNCTION_2	PUT_X_VALUE	U2_XVAR_XVAR	U2_XLVAL_XVAR
GET_LIST	U2_XLVAL_XVAR	UNIFY_X_VARIABLE	FIRSTCALL
UNIFY_X_LOCAL_VALUE	GET_STRUCTURE	FIRSTCALL	LASTCALL
UNIFY_X_VALUE	FIRSTCALL	GET_A1_VARIABLE_XN	U2_XVAR_XVAR
GET_STRUCTURE	LASTCALL	GET_A0_VARIABLE_XN	GET_A0_VARIABLE_XN
PROCEED	FUNCTION_2_IMM	GET_A0_VARIABLE_X1	FUNCTION_2_IMM
ALLOCATE	GET_YVAR_YVAR	GET_A2_VARIABLE_XN	GET_YVAR_YVAR
FIRSTCALL	UNIFY_X_VARIABLE	GET_A0_VARIABLE_X2	TRY
DEALLOCATE	TRY	FUNCTION_2_IMM	GET_A3_VARIABLE_X2
FUNCTION_2_IMM	PUT_STRUCTURE	UNIFY_LOCAL_VALUE_X1	UNIFY_VARS_XN_X2
GET_X_VALUE	PUT_Y_VALUE	UNIFY_LOCAL_VALUE_X3	PUT_STRUCTURE
TRY	GET_STRUCTURE_XVAR_XVAR	PROCEED	PUT_Y_VALUE
PUT_STRUCTURE	FAIL	TRY	GET_STRUCTURE_XVAR_XVAR
PUT_Y_UNSAFE_VALUE	U2_XVAL_XVAL	PUT_Y_VALUE	FAIL
CUTB	PUT_CONSTANT	UNIFY_VARIABLE_X3	GET_A1_STRUCTURE
FAIL	GET_YFVAR_YVAR	GET_A3_VARIABLE_X2	U2_XVAL_XVAL
UNIFY_Y_VARIABLE	GET_X_VALUE	ALLOCATE	PUT_CONSTANT
PUT_CONSTANT	PROCEED	GET_A1_VARIABLE_X3	GET_YFVAR_YVAR
PUT_Y_VARIABLE	BUILTIN_2	UNIFY_VARS_XN_X2	GET_X_VALUE
UNIFY_VOID	PUT_Y_VARIABLE	GET_A3_VARIABLE_XN	PROCEED
BUILTIN_2	BUILTIN_2_IMM	GET_Y_VARIABLE	BUILTIN_2
BUILTIN_2_IMM	GET_Y_VARIABLE	GET_STRUCTURE	PUT_Y_VARIABLE
BUILTIN_1	CUTB	GET_A1_STRUCTURE	BUILTIN_2_IMM
		FAIL	

Table 10: The most frequent instructions for the different machines. Only the 30 most frequent instructions are shown. It is worth noticing that EXECUTE gets such a dominating role in M_1, M_2 and M_3.

Machine	M_0	M_1	M_2	M_3
Number of opcodes	136	189	427	244
Emulator size on Sparc	17148	23196	36396	28248
Emulator size on x86	18316	23548	41420	31240

Table 11: Emulator sizes in bytes. The size of the main function of the emulator (the wam function) which is the one that represents the change in size for the whole emulator. The size is slightly larger on the x86 machine.

On the x86 machine the best machine is instead M_3; see Table 2. The result is not as clear as on the Sparc machine since there are different winners for different parts, but it is clear enough to show an improvement. Also M_2 performs better which leads to the conclusion that specializations are more favorable on a machine with fewer registers. The reduced register pressure is more beneficial.

The disappointing result of M_3, on the Sparc architecture Table 1, suggests that the development of a machine with more combinations is the way to go. The test done with threading disabled supports this belief, since in Table 4 the heavy specialized machines M_2 and M_3 perform worse.

9.1.2 Byte-code size

In Table 5 the size of the generated bytecode can be compared. The space savings are about 7% going from the almost WAM equivalent M_0 to M_1 (SICStus of today). When comparing M_0 to M_2 the saving is even greater, 12%. M_3 is runner up in the space-saving race, but clearly beaten by M_2. The difference between M_2 to M_3 is 5%. This means that the bytecode of M_2 is the most compact, as expected since this machine has so many opcodes.

9.1.3 Emulator size

The emulator size was measured as the size of the `wam` function in the SICStus emulator object file `wam.o` for each machine. It has different sizes for the different machines. Other parts of the emulator also differ in size, but that difference is not of interest for this work. To get a fair comparison, the emulator size of the optimized version (without debugging) was used. There is a clear correlation between emulator size and the number of opcodes in the abstract machine, as expected.

There is a higher penalty on the number of opcodes on the x86 architecture. Why is not clear and has not been investigated. The assembly code on the different architectures might help to explain. The data collected is not sufficient to make any clear conclusions, but the size difference between the Sparc and x86 does seem to increase more per opcode, the more opcodes there are. Values are given in Table 11.

The average number of bytes required for implementing new opcodes is calculated for each machine in Table 12. The highly specialized M_2 has a lower penalty per opcode. This suggests that specialized opcodes will be more compact.

9.1.4 Disassembly of some frequent predicates

Some of the benchmarks that gave unexpected results (lack of improvement or surprisingly good improvement) were investigated closer by disassembling the generated code.

Machine	M_0	M_1	M_2	M_3
Number of opcodes	136	189	427	244
Size per opcode on Sparc	126	123	85	116
Size per opcode on x86	135	125	97	128

Table 12: Average opcode size in bytes.

The partition predicate in qsort

The greatly increased performance of Quintus for the `qsort` benchmark is one example of an unexpectedly good improvement. Table 1 and Table 2 show that `qsort` executes considerably faster on M_2 than on M_0 which was expected since it contains specializations and combinations that do not exist in M_0, but the large difference was unexpected. Profiling of the `qsort` benchmark showed that it spends most of its time in a predicate called `partition`. The disassembled code of partition is shown in Table 13.

The concatenate predicate in nreverse

Another benchmark that performed very well is `nreverse`. Profiling showed that most time is spent in a predicate called `concatenate`. Disassembly of `concatenate`, for each machine is showed in Table 14. The first instruction in the table (`get_list_x0`) is actually skipped and causes no dispatch.

9.2 Space and time results

The space measurements are exact, whereas the time measurements have a stochastic pattern. There are some dependences between the three measured units. A smaller code size often require a larger emulator with many combinations. A larger emulator might run slower, evaluate one instruction, but usually each instruction is now potentially several simple instructions and as a result each basic instructions might be carried out quicker. Because of their dependence, execution times and memory usage have to be compared in parallel to give the whole picture.

The size of the emulator itself seems to be of limited importance for most applications, since it only varies slightly. Although sometimes inaccurate the time measurements can often give the most important information. This was a very frustrating situation. Time measurements varying much more than ten times the uncertainty of the measurements, but still being the most valuable measurement. How can this be dealt with? The first thing to remember is that this was a fact, and all conclusions drawn from these measurements must be treated with caution. The second thing to remember is the fundamentals of statistics, more measurements, less uncertainty.

This thesis also clearly shows that it is wrong to rely solely on small benchmarks, as they do not reflect the correct potential of improvements on larger, real world, benchmarks. For size measurements they are satisfactory see Table 5. That the correlations for time measurements is less clear can be seen in Table 1 and Table 2.

9.3 Recommendations

9.3.1 Worthwhile?

Improvements have in this thesis also been proven to be easily achieved and worthwhile. The size penalty on the emulator is not large, see Table 11 and no other real

M_0		M_2	
GET_LIST_X0		GET_LIST_X0	
UNIFY_X_VARIABLE[x(4)]		UNIFY_VARS_XN_X0[x(4)]	c
UNIFY_X_VARIABLE[x(0)]			
GET_LIST[x(2)]		GET_A2_LIST	s
UNIFY_X_VALUE[x(4)]		UNIFY_X_VALUE[x(4)]	
UNIFY_X_VARIABLE[x(2)]		UNIFY_VARIABLE_X2	s
HEAPMARGIN_CALL[3,5 live]		HEAPMARGIN_CALL[3,5 live]	
BUILTIN_2[<builtin 0xff2e8140>,x(4),x(1) else fail]		BUILTIN_2[<builtin 0xff22bc2c>,x(4),x(1) else fail]	
CUTB		CUTB	
EXECUTE[user:partition/4]		EXECUTE[user:partition/4]	
GET_LIST_X0		GET_LIST_X0	
UNIFY_X_VARIABLE[x(4)]		UNIFY_VARS_XN_X0[x(4)]	c
UNIFY_X_VARIABLE[x(0)]			
GET_LIST[x(3)]		GET_A3_LIST	s
UNIFY_X_VALUE[x(4)]		UNIFY_X_VALUE[x(4)]	
UNIFY_X_VARIABLE[x(3)]		UNIFY_VARIABLE_X3	
EXECUTE[user:partition/4]		EXECUTE[user:partition/4]	
Base case, not so interesting since not much time is spent here.			
GET_NIL_X0		GET_NIL_X0	
GET_NIL[x(2)]		GET_A2_NIL	s
GET_NIL[x(3)]		GET_A3_NIL	s
PROCEED		PROCEED	

Table 13: The predicate `partition` disassembled. Results for M_0 and M_2 is shown. c stands for combination and s for specialization. If an opcode takes parameters, then they are given within square brackets.

M_0		M_1		M_2		M_3	
GET_LIST_X0		GET_LIST_X0		GET_LIST_X0		GET_LIST_X0	
UNIFY_X_VARIABLE [x(3)]		U2_XVAR_XVAR [x(0),x(3)]	C	UNIFY_VARS_X3_X0	C+S	UNIFY_VARS_X3_X0	C+S
UNIFY_X_VARIABLE [x(0)]							
GET_LIST [x(2)]		GET_LIST [x(2)]		GET_A2_LIST	S	GET_A2_LIST	S
UNIFY_X_VALUE [x(3)]		U2_XVAL_XVAR [x(2),x(3)]	C	UNIFY_VALUE_X3	S	U2_XVAL_XVAR [x(2),x(3)]	C
UNIFY_X_VARIABLE [x(2)]				UNIFY_VARIABLE_X2	S		
EXECUTE [user:concatenate/3]		EXECUTE [user:concatenate/3]		EXECUTE [user:concatenate/3]		EXECUTEQ [user:concatenate/3]	
GET_NIL_X0		GET_NIL_X0		GET_NIL_X0		GET_NIL_X0	
GET_X_VALUE [x(1),x(2)]		GET_X_VALUE_PROCEED [x(1),x(2)]	C	GET_A1_VALUE_X2	S	GET_X_VALUE_PROCEED [x(1),x(2)]	C
PROCEED				PROCEED			
Opcodes	10		7		9		7
Combinations	0		3		1		3
Specializations	0		0		5		2
Dispatches for each run of the inner loop	6		4		5		4
Operand decodes per inner loop run	6		6		1		3
Execution time of **nreverse** (M_x/M_0), on x86	1.00		0.75		0.84		0.74
Execution time of **nreverse** (M_x/M_0), on Sparc	1.00		0.75		0.63		0.65
Execution time of **nreverse** (M_x/M_0), average	1.00		0.75		0.73		0.70

Table 14: The predicate `concatenate` disassembled. Results for all machines are shown. c stands for combination and s for specialization. If an opcode takes parameters, then they are given within square brackets.

obstacles for incrementing the instruction set was encountered. The recommendations for achieving improvements of around 10% would be to follow the outline of this thesis. Looking at pairs of instructions to find good mergers and looking at the instruction frequencies to get good candidates for specializations. On a x86 architecture all these improvements will be beneficial. On a Sparc architecture the pairs and the combinations would pay of the most.

Remember to make sure that any introduced opcodes actually come to use. This can easily be done by examining whether the instruction counts are zero or not. Zero counts obviously imply that the optimizations were not used. This might be due to missing translations rules or wrong ordering of specializations and combinations, where these are exclusive.

9.3.2 Sparc versus x86

There are some distinct differences on the recommendations one can give for implementations on Sparc versus x86. The conclusions that has been supported by this thesis is primarily the value of specializations versus combinations.

For implementations on a x86 architecture the value of specializations seem to be better than on the Sparc or Sparc-like architecture with many registers. The difficulty of register allocations seem to more easily result in improvements on architecture with less registers. The more information, on which registers will be used, the better on such an architecture. The same should be true for the Sparc architecture, but the impact is much smaller since there are so many more registers available.

9.3.3 Combinations versus specializations

As one might be able to see in the tables a very frequent instruction that is combined will drop in frequency. That is what is expected since it is combined some of the times when it occurs and therefore does not occur so often on its own in the counts. This means that one has to choose between combining and specializing since once one is done the frequency of the instructions will drop and it will not be the best choice for improvement once one of the two is done.

The best choice depends on the improvements required. If speed is needed, go for combinations. If compact code is of the essence start with specializations.

9.4 Improvements for a SICStus similar machine

Here is a short description on how to implement the optimal machine which was not implemented during this thesis work.

The value of existing optimizations in SICStus can be seen by comparing M_0 and M_1 in Table 1 and 2. Apart from the improvements already in place in SICStus the following steps can be taken to improve the performance of SICStus virtual machine.

- Remove the opcode `put_x_value` and `put_xval_xval` and let them be translated into `get_x_variable` and `get_xvar_xvar`. Any combinations and specializations will now be beneficial for both opcodes. The order of which combinations are applied will be more important, best is to implement this improvements and then look at pairs and frequencies of opcodes.

- Combine the opcode pairs marked with c in table 7.

- Specialize the opcodes pairs marked with s in table 7.
- Rerun the benchmarks suite and collect new data.

Two or three runs of collecting data, optimizing and comparing is probably enough to improve the machines performance by at least 10%. It has been shown to be true for code size, and a 2% performance on runtime on the x86 machine was shown using only a few improvements.

9.5 Future work

With all goals satisfied, it would be interesting to look into an alternative compilation of arithmetic to reduce dereferencing and tagging. Inline compilation of disjunctions is another area for improving SICStus.

Some work has been done in the area of dynamic optimization instead of static optimization [11]. Static and dynamic optimizations differ in the way that static optimizations can never be optimal for all programs. The goal is instead to find an optimal set that is "on the average" optimal. In a dynamic approach an optimization for a *particular* program is sought. This results in applications for dynamic optimization differing slightly from the ones for static. The dynamic optimizations are to be used where enough time and effort can be spared to do a separate optimization for each program. The optimal solution would be to first find a very good instruction set with static optimizations and then work on the speed of the dynamic optimizations. To find dynamic optimization ([10]) before the static one has been introduced is not economic.

Some of the optimizations discussed in this thesis are applicable to all abstract machine while others are Prolog, or even SICStus specific.

9.6 If only there were more time

With more time and resources it would be possible to build a completely new compiler, and build it around the abstract machine. Building a new compiler might be too large a quest, but thought in that direction might lead to new ideas and improvements to existing compilers.

One would like to see a compiler where the abstract machine is more easily changeable, possibly a higher level of abstraction where it is easier to modify and evaluate different machine configurations on an equal basis. To create such an abstract foundation could lead to finding a perfect and optimal machine for all given situations. Such information could then be implemented into existing systems. The higher level of abstraction would also give more room for parallel development of independent areas of the machine.

The most challenging area would be to work on the inter-procedural pairs, that only were counted here.

10 Conclusions

Abstract machines can be improved by different methods. One way is to expand the instruction set to include instructions specialized for certain registers. Another way is to include instructions that are combinations of several simpler instructions (i.e., combinations). This thesis shows that these can be worthwhile tasks slightly favoring

combinations. It is also shown that there are some significant differences in improvement between different platforms such as Sparc and x86. The technique is of more use on an architecture with few registers.

Several different abstract machines were implemented to find out how much SICStus Prolog could benefit from different improvements as well as to evaluate how different optimizations pay off. Non threaded versions were also compared to the threaded versions.

Both large and small benchmarks were used to see how the techniques scale and whether small benchmarks can be used as predictions of how large programs will behave. The results show that is unsafe to solely rely on small benchmarks.

Execution time, emulator size and code size were measured. Improvements of the order of 10% in time benchmarks and at least improvements of the order of 15% in the code size with small penalties on the emulator size were observed.

11 Acknowledgments

I would like to mention and thank; supervisor Dr. Mats Carlsson, examiner Associate Professor Konstantinos Sagonas, Rosemary Rothwell and Cédric Cano for their contributions to this thesis.

Stimulating seminars and discussions with Bart Demoen and Richard O'Keefe also served to increase the motivation for this thesis. The tutorial by Hassan Aït-Kaci has also been of great help writing this thesis.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[3] Karen Appelby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for prolog based on wam. *Communications of the ACM*, 31(6):719–741, February 1988.

[4] James R. Bell. Threaded code. *Communications of the ACM*, 16(8):370–373, June 1973.

[5] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog, 4th edition*. Springer Verlag, Berlin, Germany, 1994.

[6] Bart Demoen and Phuong-Lan Nguyen. So many WAM variations, so little time. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of Computational Logic — CL-2000*, number 1861 in LNAI, pages 1240–1254. Springer, July 2000.

[7] M. Anton Ertl and David Gregg. Hardware support for efficient interpreters: Fast indirect branches. Unpublished technical report, 2000.

[8] Mats Carlsson et al. SICStus Prolog user's manual. ISBN 91-630-3648-7, Swedish Institute of Computer Science, 1995.

[9] Paul Klint. Interpretation techniques. *Software Practice and Experience*, 11(9):963–973, September 1981.

[10] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental. *IEEE International Conference on Computer Languages (ICCL '98)*, Chicago, Illinois, USA:123–142, May 1998.

[11] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300, 1998.

[12] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, January 1995.

[13] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, and Ernie Johnson. *The XSB System, Version 2.2. Volume 1: Programmer's Manual*, June 2000. See also: http://www.cs.sunysb.edu/~sbprolog.

[14] Vítor Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[15] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[16] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.

[17] Patrick Weemeeuw and Bart Demoen. Garbage collection in Aurora: An overview. In Yves Bekkers and Jaques Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 454–472. Springer-Verlag, September 1992.

Appendix A: Warren Abstract Machine

Introduction

This is a short description of the Warren Abstract Machine (**WAM**) as first presented by D. H. Warren in his report, 1983. Many Prolog bytecode emulators, among which SICStus is one, have used the WAM as their foundation. Since understanding of the WAM is vital to anyone trying to improve its instruction set, some information about it is included here in this appendix.

Data objects, data area, registers, variables and instruction set are terms used in many texts about the WAM. Some of these terms are described below. To ease the use of this appendix, important terms are in bold face when they occur close to their definitions.

Memory

The memory contains five areas, four stacks and the code area with the program. The four stacks are the heap, the stack, the trail and, the smallest one, the Push-Down-List (PDL).

Each area has pointers (registers with memory addresses) to track execution. The registers (global pointers) can be updated by instructions. Choice-points and environments are created to support the flow of control and maintain local variables throughout the execution.

Heap (Global stack)

The HEAP has the H register pointing to the top of the heap, S pointing to the next term argument to be investigated on a unification. The HB register caches the value of H, at the time of the creation of the latest choice point.

Stack (Local stack)

B points to the latest (chronologically) choice point, and E points to the current environment in the STACK.

Trail

During execution variables might be bound to terms. Sometimes these variables might need to be made unbound to enable other possibilities upon backtracking, these variables are called conditional variables.

The TRAIL contains all conditional variables from both stack and heap that have been bound. (Variables that might be discarded upon backtracking.) Such variables need to be reset upon backtracking beyond the corresponding choice point. TR points to the top of the TRAIL.

PDL

The Push Down List (PDL) is used for recursive operations, mainly unifications. The terms to be unified are built on the PDL.

Code-area

P and CP are the pointers into the code area. P is the program pointer. CP is the continuation pointer.

Memory structure

The data area is used for storage of data objects. Data objects consist of a value and a tag. The tag is one of: references, structures, **lists** and **constants**. **References** represent variables. An unbound variable references itself. A **structure** denotes that the value is the name and the arity of a functor.

The data objects are stored in memory in a number of stacks. The data areas or stacks "making up" the memory are; the code area, the heap, the stack, the trail and the Push Down List (PDL). The **heap** is used as a global stack and contains all structures and lists created by procedure calls and unifications. The **PDL** is used for unifications. The **trail** has a list of all the variables that have been bound, and that need to be unbound on backtracking.The **stack** contains environments and choice points.

Environments contain variables and their values as they occur in the body of some clause. An environment also contain a pointer into another clause. This continuation pointer represents a list of instantiated goals that have not been executed. **Choice points** contain all information needed to restore an earlier state upon backtracking.

The demands the machine puts on the structure of the code are that the heap is on lower addresses than the stack and both grow to higher addresses, (or the stack lower than the heap and growing towards lower addresses.) This organization prevents dangling references when consistently binding the variable with the highest address.

The structure used in the original WAM has the code area residing on the lowest memory addresses and it grows to higher addresses. After the code, the heap, the stack and the trail follows, all growing towards higher addresses. The PDL resides on the highest memory address and grows towards lower addresses.

Registers

The WAM is register based and registers A1, A2,...,A*n* are available as argument registers. They are used to pass arguments to procedures. *X1*, *X2*,...,*Xn* are used to store temporary variables. Implementation wise, Ai and Xi are actually the same register. Registers are also used for pointers to particular places in the memory. Together they constitute the registers of the virtual machine.

The registers determine which state the calculations are in.

- P, CP - Program pointer (P) and Continuation Pointer (CP) both to the code area.
- E, B, A - last Environment (E), Backtrack point (B) i.e., the last choicepoint, top of stack (A) all directed into the stack.
- TR, H, HB - top of TRail (TR), top of Heap (H). Heap Backtrack point (HB), the value of register H at the time of the latest choicepoint.

- S -Structure pointer to the heap (S)
- A1, A2, A3, ..., An - Argument registers, used to pass arguments to procedures.
- X1, X2, X3, ..., Xm - Temporary variable[4]registers, used to store the values of clauses' temporary variables.

The registers H, TR, B, CP, E, A, A1 through Ai and a pointer to alternative clauses has to be store for each choicepoint.

Instructions

The WAM is comprised of 42 instructions. They are all briefly presented below. Y_n represents a permanent variable, i.e., not an temporary one. C is a constant and F is a functor.

Get instructions

- `get_variable` Y_n, A_i - Assigns Y_n the value of A_i. Used for head arguments that are unbound variables.
- `get_variable` X_n, A_i - Assigns X_n the value of A_i. Used for head arguments that are unbound variables.
- `get_value` Y_n, A_i - Unifies Y_n with A_i. Used for head arguments that are bound variables.
- `get_value` X_n, A_i - Unifies X_n with A_i. Used for head arguments that are bound variables.
- `get_constant` C, A_i - Unifies a constant with A_i for head arguments that are constants.
- `get_nil` A_i - Unifies the constant [] with A_i. Used for head arguments that is the [] constant.
- `get_structure` F, A_i - Unifies A_i with a structure. Used when the head argument is a structure.
- `get_list` A_i - Unifies A_i with a list. Used when the head argument is a list.

Put instructions

- `put_variable` Y_n, A_i - Assigns Y_n and A_i a new variable. Used for goal arguments that are unbound permanent variables.
- `put_variable` X_n, A_i - Assigns X_n and A_i a new variable. Used for final goal arguments that are unbound variables.

[4]In Warren's own words: "A temporary variable is a variable that has its first occurrence in the head or in a structure or in the last goal, and that does not occur in more than one goal in the body, where the head of the clause is counted as part of the first goal. Temporary variables do not need to be stored in the clause's environment."

- `put_value` Y_n, A_i - Puts the value of Y_n into A_i. Used for goal arguments that are bound variables.
- `put_value` X_n, A_i - Puts the value of X_n into A_i. Used for goal arguments that are bound variables.
- `put_unsafe_value` Y_n, A_i - Puts the value of Y_n into A_i. Used for the last occurrence of an unsafe variable.
- `put_constant` C, A_i - Puts the constant into A_i. Used for goal arguments that are constants.
- `put_nil` A_i - Puts the constant [] into A_i. Used for goal arguments that is the [] constant.
- `put_structure` F, A_i - Assigns A_i the structure F. Used when the goal argument is a structure.
- `put_list` A_i - Assigns A_i a list. Used when the goal argument is a list.

The instruction `put_unsafe_value` replaces `put_value` in the last goal where the unsafe variable[5] is used. It ensures that the variable, if needed, is stored on the heap, and thus globalized.

Unify instructions

Unify instructions both unify existing structures and create new ones.

- `unify_void` N - Unifies N single occurring variables when they appear as head structure arguments.
- `unify_variable` Y_n - Unifies Y_n with the next subterm, using a new variable. Used for head structure arguments.
- `unify_variable` X_n - Unifies X_n with the next subterm, using a new variable. Used for head structure arguments.
- `unify_value` Y_n - Unifies Y_n with the next subterm. Used for head structure arguments.
- `unify_value` X_n - Unifies X_n with the next subterm. Used for head structure arguments.
- `unify_local_value` Y_n - Unifies Y_n with the next subterm, using a new variable. Used for head structure arguments that is not necessarily global.
- `unify_local_value` X_n - Unifies X_n with the next subterm, using a new variable. Used for head structure arguments that is not necessarily global.
- `unify_constant` C - Unifies a constant with the next subterm. Used for head structure arguments.

[5] In Warren's own words: "An unsafe variable is a permanent variable that did not first occur in the head or in a structure, i.e, the variable was initialized by a put_variable instruction."

- `unify_nil` - Unifies the [] constant with the next subterm. Used for head structure arguments.

If the variable has not been initialized by `unify_variable`, the instruction `unify_local_value` is used in the place of `unify_value`.

Procedural instructions

Allocation of environments and control flow is handled by the procedural instructions. P is a predicate and N the number of variables.

- `proceed` - Terminates a clause and sets P to CP.
- `allocate` - Beginning of a clause with more than one goal, creates an environment.
- `execute` P - Terminates last goal, P is set to the procedure.
- `deallocate` - Before final `execute` when more than one goal in the body, discards an environment.
- `call` P, N - Terminates a non-last body clause, sets CP to the following code and P to the procedure.

Indexing instructions

- `try_me_else` L - Creates a choicepoint with alternative label L. Used before code of first clause in procedures with more than one clause.
- `try` L - Creates a choicepoint with alternative label set to the next instruction. Proceeds to label L.
- `retry_me_else` L - L replaces the alternative label. Used before code of any clause (not the first and last) in procedures with more than one clause.
- `retry` L - The next instruction replaces the alternative label. Proceeds to choicepoint L.
- `trust_me_else` $fail$ - Discards the latest choicepoint.
- `trust` L - Discards the latest choicepoint. Proceeds to label L.
- `switch_on_term` L_v, L_c, L_l, L_s - P is set to one of the four arguments depending on A1's dereferenced type (variable, constant, list or structure). Used when non variable in first head argument.
- `switch_on_constant` N, $Table$ - Searches the table for a given key. The key being the constant found in A1. If the clause is found then P is set to the corresponding clause, otherwise it fails. Used for first head argument.
- `switch_on_structure` N, $Table$ - Same effect as `switch_on_constant`, but uses principal functor of A1 as key.

L, L_v, L_c, L_l, L_s are addresses of clauses and $Table$ is a Hash table of size N.

Basic Operations

Two other basic operators mentioned in the WAM are *fail* and *trail(R)*. *Fail* is used when unification fails and another clause must be tried. The *trail(R)* operation is used to (for unification) bind a variable with reference R.

Sources

There are more complete descriptions of the WAM available. This appendix is only meant to serve as a quick reference. Warren's original report [16] and Aït-Kaci's tutorial reconstruction [2] are recommended texts for the interested reader. A concise instruction description is given in [3].

Appendix B: SICStus instruction set

Abstract instructions

In this thesis, an abstract distinction is in made between abstract instructions and operation codes. The instructions of an abstract machine are called abstract instructions. The abstract instructions in this report are the same for all machines considered. The instructions are the SICStus instructions. They are more or less the same as the WAM instructions with added support for the cut operation, arithmetics, memory management, etc.

The operation codes, or opcodes, change from one machine to another as different optimizations are introduced.

Techniques used in SICStus abstract machine

Alignment issues (q)

Since some code end on a halfword (unaligned) the next operation code might or might not be unaligned in memory. Since long (full word) operands need to be aligned, this implementation gives rise to the need for having both an aligned and an unaligned version of each opcode. As a naming convention all unaligned versions end with a q, and the unaligned version simply moves the pointer P forward one halfword and then calls the aligned version of the opcode. The aligned version can then assume that all its long operands are aligned. Below in the description of opcodes only the q version is presented and it is assumed obvious that there exists an aligned version for each unaligned one. Opcodes with no trailing q do not have any long operands and can exist both aligned and unaligned in the code.

Modules (_module)

A module is a part of a program kept separate from the rest of the code. To support the use of different modules most operation codes have a duplicate. By convention these opcode names contain the word module.

Initialization of permanent variables

When garbage collection is introduced into the WAM, the problem of deciding which permanent variables contain valid terms must be solved. The solution chosen in the SICStus WAM is to always initialize all permanent variables before the first body call, and to never re-initialize them after the first body call. This implies that the instructions `get_y_variable`, `put_y_variable`, and `unify_y_variable` must be replaced by `get_y_value`, `put_y_value`, and `unify_y_value` respectively after the first body call. Also, a new instruction explicitly initializes permanent variables.

There is a similar decision problem with temporary variables. The SICStus WAM solves this problem by only admitting garbage collection in contexts where the set of live temporary variables is known.

Nop (_void)

Some instructions exploit the fact that instructions move registers onto themselves. The _void opcodes are specializations for the nop case.

Indexing (_x0)

Indexing is supported through the opcodes ending with x0. In the WAM the switch instructions `switch_on_term`, `switch_on_constant` and `switch_on_structure` handle indexing. The `switch_on_term` instruction skip to different positions depending on the type of A1. `switch_on_constant` and `switch_on_structure` jump to different positions depending on the principal functor of A1. These instructions are not used in SICStus.

In SICStus indexing is handled by `call` and `execute`. Depending on which type of predicate these instructions call slightly different things are done. Indexing is done according to a pattern when the type of the predicate is "indexed".

```
switch (type_of(A1)) {
        case variable: ...
        case list: ...
        default: switch (functor_of(A1)) {
        case foo/0: ...
        case bar/1: ...
        default: ...
        }
}
```

For the list-case the code following usually starts with `get_list_x0`. In that case the instruction is a no-op and the compiler back-end ensures that the list-case branches to the following instruction. The same thing applies to the constant-case (`get_nil_x0` or `get_constant_x0`) and the structure-case (`get_structure_x0`), these _x0 instructions also becomes no-ops and can be skipped. For any other case (after backtracking or if A1 is a variable) the _x0 instructions have to be executed.

Extended opcode set

Combinations and specializations are extensively used in the machines to extend and improve the opcode set. A specialization is an abstract instruction that has been split into several opcodes. Each new opcode deals with a special case. Usually there is also an opcode to catch all other cases, i.e., put_variable X can be specialized resulting in put_X0, put_X1, put_Xn. (They do not have to be sequential, but mostly they are.) Extensively used in Quintus, but rarely in SICStus to date.

A combination is several instructions merged into a single opcode.

In some cases all possible combinations can be created and the original instruction removed, i.e., allocate and deallocate. Combinations include the var_var opcodes.

Bignums (_large)

Large numbers are supported. Operation codes containing **large** deal with those issues that arise for large integers and floating point numbers.

Arithmetics

Support for functions, comparisons, arithmetics, profiling etc. Some opcodes take an immediate operand and the name then contain **_imm**.

Foreign language interface

Opcodes for foreign language (FLI) support are included in the machine, but they are for simplicity (they are not relevant for this work) left out of this appendix.

The SICStus instruction set

The opcodes used to implement SICStus are built on the WAM instructions, but have also been extended. SICStus 3.8 abstract machine also contains optimizations. All optimizations where removed from SICStus 3.8 to get an instruction set as similar to the basic WAM as possible as a starting point.

`Allocate` and `deallocate` were also reintroduced to make it possible to remove combinations containing them. The resulting machine is called M_0. The instructions used to implement M_0 is what in this report is called SICStus instruction set.

Opcodes marked by an asterisk represent zero or more occurrences of that opcode.

A compact yet easy to understand syntax has been used. Each combination or specialization is described in terms of basic SICStus instructions. The opcode is to the left and an implication arrow shows which basic instructions it is created from.

Instructions with no similar instruction in the WAM are in capital letters. Existing WAM instructions are not in capital letters. The instructions have been loosely grouped together.

Procedural instructions

- `INIT` $List$: Initializes permanent variables. $List$ represent a list of the variables.
- `allocate`: Beginning of a clause with more than one goal, creates an environment.
- `deallocate`: Before final `execute` when more than one goal in the body, discards an environment.
- `call` P, S: Terminates a non-last body clause, sets CP to point to the next instruction and P to the procedure.
- `CALL_IN_MODULE` P, S: A `call` version for use in modules.

Initializes permanent variables and performs a call.

- $\texttt{FIRSTCALLQ(List,P,S)} \Leftarrow \texttt{INIT(List)}, \texttt{call(P,S)}$

Initializes permanent variables and performs a call in modules.

- $\texttt{FIRSTCALL\_IN\_MODULEQ(List,P,S)} \Leftarrow \texttt{INIT(List)}, \texttt{CALL\_IN\_MODULE(P,S)}$
- `executeq` P: Terminates last goal, P is set to the procedure.
- `EXECUTE_IN_MODULEQ` P: An `execute` version for use in modules.
- `proceed`: Terminates a clause and sets P to CP.

PUT instructions

- `put_x_variable` X_n, A_i: Assigns X_n and A_i a new variable. Used for final goal arguments that are unbound variables.
- `PUT_X_VOID` A_i: Specialized `put_x_variable` for the cases when the two registers (A_i, X_i) it operates on are the same.
- `put_x_value` X_n, A_i: Puts the value of X_n into A_i. Used for goal arguments that are bound variables.
- `put_y_variable` Y_n, A_i: Assigns Y_n and A_i a new variable. Used for goal arguments that are unbound permanent variables.
- `put_y_value` Y_n, A_i: Puts the value of Y_n into A_i. Used for goal arguments that are bound variables.
- `put_y_unsafe_value` Y_n, A_i: Puts the value of Y_n into A_i. Used for the last occurrence of an unsafe variable.
- `put_constantq` C, A_i: Puts the constant into A_i. Used for goal arguments that are constants.
- `put_structureq` F, A_i: Assigns A_i the structure F. Used when the goal argument is a structure.
- `put_nil` A_i: Puts the constant [] into A_i. Used for goal arguments that is the [] constant.
- `put_list` A_i: Assigns A_i a list. Used when the goal argument is a list.
- `PUT_LARGEQ` C, A_i: Version of `PUT_CONSTANTQ` for floating point numbers and large integers. Puts a reference to the bignum into A_i.

Get instructions

- `get_x_variable` X_n, A_i: Assigns X_n the value of A_i. Used for head arguments that are unbound variables.
- `get_y_variable` Y_n, A_i: Assigns Y_n the value of A_i. Used for head arguments that are unbound variables.
- `get_x_value` X_n, A_i: Unifies X_n with A_i. Used for head arguments that are bound variables.
- `get_y_value` Y_n, A_i: Unifies Y_n with A_i. Used for head arguments that are bound variables.
- `GET_Y_FIRST_VALUE`$(\mathtt{Y_n}, \mathtt{A_i}) \Leftarrow$ `get_y_variable`$(\mathtt{Y_n}, \mathtt{A_i})$ after the first body call.
- `get_constantq` C, A_i: Unifies a constant with A_i for head arguments that are constants.
- `GET_CONSTANT_XOQ` C : Specialized `get_constant` for indexing on A1.

- `get_structureq` F, A_i: Unifies A_i with a structure. Used when the head argument is a structure.
- `GET_STRUCTURE_X0Q` F: Specialized `get_structureq` for indexing on A1.
- `get_nil` A_i: Unifies the constant [] with A_i. Used for head arguments that is the [] constant.
- `GET_NIL_X0`: Specialized `get_nil` for indexing on A1.
- `get_list` A_i: Unifies A_i with a list. Used when the head argument is a list.
- `GET_LIST_X0`: Specialized `get_list` for indexing on A1.
- `GET_LARGEQ` C, A_i: Specialized `get_constant` for floating point numbers and large integers.
- `GET_LARGE_X0Q` C: Version of `GET_LARGEQ` for indexing on A1.

Unify instructions

- `unify_void` N: Unifies N single occurring variables when they appear as head structure arguments.
- `unify_x_variable` X_n: Unifies X_n with the next subterm, using a new variable. Used for head structure arguments.
- `unify_y_variable` Y_n: Unifies Y_n with the next subterm, using a new variable. Used for head structure arguments.
- `unify_x_value` X_n: Unifies X_n with the next subterm. Used for head structure arguments.
- `unify_x_local_value` X_n[6]: Unifies X_n with the next subterm, using a new variable. Used for head structure arguments that are not necessarily global.
- `unify_y_value` Y_n: Unifies Y_n with the next subterm. Used for head structure arguments.
- `unify_y_local_value` Y_n[7]: Unifies Y_n with the next subterm, using a new variable. Used for head structure arguments that are not necessarily global.
- `UNIFY_Y_FIRST_VALUE`$(\mathtt{Y_n}) \Leftarrow$ `unify_y_variable`$(\mathtt{Y_n})$ after the first body call.
- `UNIFY_CONSTANTQ` C: Unifies a constant with the next subterm. Used for head structure arguments.
- `UNIFY_STRUCTUREQ`$(\mathtt{X_n}, \mathtt{F}) \Leftarrow$ `unify_x_variable`$(\mathtt{X_n})$, `get_structure`$(\mathtt{F}, \mathtt{A_i})$
- `unify_nil`: Unifies the [] constant with the next subterm. Used for head structure arguments.
- `UNIFY_LIST`$(\mathtt{X_n}, \mathtt{A_i}) \Leftarrow$ `unify_x_variable`$(\mathtt{X_n})$, `get_list`$(\mathtt{A_i})$
- `UNIFY_LARGEQ` C: Version of `unify_constant` for floating point numbers and large integers.

[6] Used if the variable in `unify_x_variable` has not been initialized.
[7] Used if the variable in `unify_y_variable` has not been initialized.

Function opcodes (Arithmetic opcodes, addition and subtraction etc.)

For these opcodes, the value is assigned to X_n whereas A_i, A_j, A_k are input arguments.

- `FUNCTION_0Q` X_n: Zero-ary functions.
- `FUNCTION_1Q` X_n, A_i: Unary functions, such as unary minus.
- `FUNCTION_2Q` X_n, A_i, A_j: Binary functions, such as binary addition.
- `FUNCTION_2_IMMQ` X_n, A_i, A_j: This version of the `FUNCTION_2Q` opcode takes an immediate operand.
- `FUNCTION_3Q` X_n, A_i, A_j, A_k: Ternary functions.

Builtin opcodes

Used for inlined predicates such as "=..", "==", "\==", "=:=", ">" etc. For these opcodes, A_i, A_j, A_k are input arguments.

- `BUILTIN_1Q` A_i: Unary predicates.
- `BUILTIN_2Q` A_i, A_j: Binary predicates.
- `BUILTIN_2_IMMQ` A_i, A_j: This version of the `BUILTIN_2Q` opcode takes an immediate operand.
- `BUILTIN_3Q` A_i, A_j, A_k: Ternary predicates.

Other opcodes

For these opcodes, B_0 denotes B at entry to the current predicate.

- `CUTB`: Cut to B_0.
- `CUTB_X` X_n: Cut to X_n.
- `CUT_Y` Y_n: Cut to Y_n.
- `CHOICE_X` X_n: Store B_0 in X_n.
- `CHOICE_Y` Y_n: Store B_0 in Y_n.
- `KONTINUE`: Return to a procedure call after event handling.
- `LEAVE`: Return to native code.
- `EXIT_TOPLEVEL`: Exit from Abstract Machine.
- `try` L: Creates a choicepoint with alternative label set to the next alternative. Proceeds to label L.
- `RETRY_C`Q: Backtrack into C code.
- `RETRY_NATIVE`Q: Backtrack into native code.
- `RETRY_INSTANCE`: Backtrack into interpreted code.

- `HEAPMARGIN_CALLQ`: Checks for events such as stack overflows, interrupts or woken goals.
- `BUMP_COUNTERQ`: Profiling.
- `BUMP_ENTRYQ`: Profiling.
- `fail`: Used when unification fails and another clause must be tried.
- `NOP`: No operation.

Appendix C Opcodes of the 4 machines

This appendix contains descriptions of all the opcodes used in each of the four machines. A compact yet easy to understand syntax has been used. Each combination or specialization is described in terms of basic SICStus instructions. The opcode is to the left and an implication arrow shows which basic instructions it is created from, i.e.,:

`PUT_Y_FIRST_VARIABLE(Y,X)` $\Leftarrow$ `allocate,put_y_variable(Y,X)`

General register that has to be fetched to be determined are given as A, B, C, D, X or Y (also indexed version of those names are used.) To shorten the notation the following is introduced. Arguments that the opcodes take are given within parenthesis.

α is uses to signify several opcodes, one for each number α can be, that is the set $\{0, 1, 2, 3\}$. For multiple register opcodes the shorthand notation works as follows . When α and β are used they symbolize multiple opcodes where each permutation of $(\alpha,\ \beta)$ with $\alpha = \{0, 1, 2, 3\}$, $\beta = \{0, 1, 2, 3\}$. When permutations $\alpha = \beta$ of the given opcode were not created the notation $(\alpha_2,\ \beta_2)$ is used instead. This is the case when both registers are the same, neither is a Y register. γ is used when each permutation of the set $\{1, 2, 3, 4\}$ represent one opcode.

M_0's 136 opcodes

The first machine was implemented to be an as minimal SICStus abstract machine as possible while maintaining all functionality. By removing optimizations from SICStus an almost WAM equivalent machine was revealed. It is implemented with 136 opcodes. Each instruction in the WAM has been realized but support is also added for extensions. The 136 opcodes are SICStus instruction set.

M_1's 189 opcodes

M_1 is implemented with 189 opcodes. It includes all of M_0's opcodes, apart from allocate and deallocate. `Allocate` and `deallocate` have been merged with all possible trailing instructions and have been removed. The following 47 opcodes 55 (189-(136-2) counting the unaligned versions), combinations and specializations, have been added.

`Call` instructions are the same as in M_0, but can here also takes zero or more preceding put opcodes.

- `CALLQ` $\Leftarrow$ `put_value`*, `call`
- `CALL_IN_MODULEQ` $\Leftarrow$ `put_value`*, `call_in_module`

`Lastcall` (`deallocate` followed by an `execute` instruction) can here also takes zero or more preceding put opcodes.

- `LASTCALLQ` $\Leftarrow$ `put_value`*, `lastcall`
- `LASTCALL_IN_MODULEQ` $\Leftarrow$ `put_value`*, `lastcall_in_module`
- `INITCALLQ` $\Leftarrow$ `allocate`, `init`, `call`
- `INITCALL_IN_MODULEQ` $\Leftarrow$ `allocate`, `init`, `call_in_module`

Put opcodes

Allocate followed by put_y_value.

- PUT_Y_FIRST_VARIABLE(Y, X) ⇐ allocate, put_y_variable(Y, X)

Combinations of allocate and two put_y_value.

- PUT_YFVAR_YVAR(Y, X, Y1, X1) ⇐ allocate, put_y_variable(Y, X), put_y_variable(Y1, X1)
- PUT_YVAR_YVAR(Y, X, Y1, X1) ⇐ put_y_variable(Y, X), put_y_variable(Y1, X1)

Combinations of allocate and two put_y_value.

- PUT_XVAL_XVAL(A, B, C, D) ⇐ put_x_value(A, B), put_x_value(C, D)
- PUT_YVAL_YVAL(Y, X, Y1, X1) ⇐ put_y_value(Y, X), put_y_value(Y1, X1)]
- PUT_YVAL_YUVAL(Y, X, Y1, X1) ⇐ put_y_value(Y, X), put_y_unsafe_value(Y1, X1)
- PUT_YUVAL_YVAL(Y, X, Y1, X1) ⇐ put_y_unsafe_value(Y, X), put_y_value(Y1, X1
- PUT_YUVAL_YUVAL(Y, X, Y1, X1) ⇐ put_y_unsafe_value(Y, X), put_y_unsafe_value(Y1, X1)

Get opcodes

Combination of two get_x_variable.

- GET_XVAR_XVAR(X_1, A_1, X_2, A_2) ⇐ allocate, get_x_variable(X_1, A_1), get_x_variable(X_2, A_2)
- GET_YVAR_YVAR(Y_1, A_1, Y_2, A_2) ⇐ get_y_variable(Y_1, A_1), get_y_variable(Y_2, A_2)

Combination of allocate and two get_y_variable.

- GET_YFVAR_YVAR(Y_1, A_1, Y_2, A_2) ⇐ allocate, get_y_variable(Y_1, A_1), get_y_variable(Y_2, A_2)
- GET_Y_FIRST_VARIABLE(Y, A) ⇐ allocate, get_y_variable(Y, A)
- GET_CONSTANT_PROCEEDQ(A, B) ⇐ get_constant(A, B), proceed
- GET_NIL_PROCEED(A) ⇐ get_nil(A), proceed
- GET_STRUCTURE_XVAR_XVARQ(A, B, C, D) ⇐ get_structure(A, B), unify_x_variable(C),

unify_x_variable(D)

- GET_LIST_XVAR_XVAR ⇐ get_list(A, B), unify_x_variable(C), unify_x_variable(D)
- GET_CONSTANT_CONSTANTQ(A, B, C, D) ⇐ get_constant(A, B), get_constant(C, D)
- GET_X_VALUE_PROCEED(A, B) ⇐ get_x_value(A, B), proceed

Unify opcodes

- UNIFY_VOID_$\gamma \Leftarrow$ unify_void(γ)
- UNIFY_Y_FIRST_VARIABLE(X, Y) $\Leftarrow$ allocate, unify_y_variable(X)
- UNIFY_CONSTANT_PROCEED(A) $\Leftarrow$ unify_constant(A), proceed
- UNIFY_NIL_PROCEED $\Leftarrow$ unify_nil, proceed
- U2_VOID_XVAR(X, Y) $\Leftarrow$ unify_void(X), unify_x_variable(Y)
- U2_VOID_XVAL(X, Y) $\Leftarrow$ unify_void(X), unify_x_value(Y)
- U2_VOID_XLVAL(X, Y) $\Leftarrow$ unify_void(X), unify_x_local_value(Y)
- U2_XVAR_VOID(X, Y) $\Leftarrow$ unify_x_variable(X), unify_void(Y)
- U2_XVAR_XVAR(X, Y) $\Leftarrow$ unify_x_variable(X), unify_x_variable(Y)
- U2_XVAR_XVAL(X, Y) $\Leftarrow$ unify_x_variable(X), unify_x_value(Y)
- U2_XVAR_XLVAL(X, Y) $\Leftarrow$ unify_x_variable(X), unify_x_local_value(Y)
- U2_XVAL_VOID(X, Y) $\Leftarrow$ unify_x_value(X), unify_void(Y)
- U2_XVAL_XVAR(X, Y) $\Leftarrow$ unify_x_value(X), unify_x_variable(Y)
- U2_XVAL_XVAL(X, Y) $\Leftarrow$ unify_x_value(X), unify_x_value(Y)
- U2_XVAL_XLVAL(X, Y) $\Leftarrow$ unify_x_value(X), unify_x_local_value(Y)
- U2_XLVAL_VOID $\Leftarrow$ unify_x_local_value(X), unify_void(Y)
- U2_XLVAL_XVAL $\Leftarrow$ unify_x_local_value(X), unify_x_value(Y)
- U2_XLVAL_XLVAL(X, Y) $\Leftarrow$ unify_x_local_value(X), unify_x_local_value(Y)

Other opcodes

- CUTB_X_PROCEED(A) $\Leftarrow$ cutb_x(A), proceed
- CUTB_PROCEED $\Leftarrow$ cutb, proceed
- CHOICE_YF $\Leftarrow$ allocate, choice_y(Y)

M_2's 427 opcodes

M_2 implements new opcodes that corresponds to the opcodes used in Quintus Prolog. It includes all of M_0's opcodes and optimizations described below.

One type of specialization used in Quintus Prolog is a number of get_constant followed by a proceed. It is not the same optimization as the one used in SICStus since this one only is used for the case when the second register and onwards are used, i.e.,: "x1" or "x1, x2" or "x1, x2, x3". This opcode type is named get_fact and contains 6 opcodes.

- GET_FACT_1(A) ⇐ get_constant(A, 1), proceed
- GET_FACT_2(A, B) ⇐ get_constant(A, 1), get_constant(B, 2), proceed
- GET_FACT_3(A, B, C) ⇐ get_constant(A, 1), get_constant(B, 2), get_constant(C, 3), proceed

Put_constant to the first register followed by call is specialized into call_constant.

- CALL_CONSTANTQ(A, B, C) ⇐ put_constant(A, 0), call(B, C)

Put_value or put_unsafe_value followed by call are combined for the x0 register, and specialized for the four first registers (hardware registers whenever possible in Quintus).

- CALL_VALUE_YαQ(A, B) ⇐ put_y_value(α, 0), call(A, B)
- CALL_VALUE_YNQ(A, B, Y) ⇐ put_y_value(Y, 0), call(A, B)
- CALL_UNSAFE_VALUE_YαQ(A, B) ⇐ put_y_unsafe_value(α, 0), call(A, B)
- CALL_UNSAFE_VALUE_YNQ(A, B, Y) ⇐ put_y_unsafe_value(Y, 0), call(A, B)

Cut followed by proceed is, just as in M_1, combined to form cutb_proceed and cutb_x_proceed.

- CUTB_PROCEED ⇐ cutb, proceed
- CUTB_X_PROCEED(A) ⇐ cutb_x(A), proceed

Put_x_void specialized for the four first argument registers the general case put_x_void reintroduced.

- PUT_Aα_VOID ⇐ put_x_void(α)

The put_constant specializations:

- PUT_Aα_CONSTANTQ(B) ⇐ put_constant(B, α)

The put_structure specializations:

- PUT_Aα_STRUCTUREQ(B) ⇐ put_structure(B, α)

The put_nil specializations:

- PUT_Aα_NIL ⇐ put_nil(α)

The put_list specializations:

- PUT_Aα_LIST ⇐ put_list(α)

The get_y_value specializations:

- GET_Aα_VALUE_YN(A) ⇐ get_y_value(A, α)

The get_constant specializations:

- GET_Aα_CONSTANTQ(A) ⇐ get_constant(A, α)

The get_structure specializations:

- GET_Aα_STRUCTUREQ(A) $\Leftarrow$ get_structure(A, α)

The get_nil specializations:

- GET_Aα_NIL $\Leftarrow$ get_nil(α)

The get_list specializations:

- GET_Aα_LIST $\Leftarrow$ get_list(α)

The unify_void specializations:

- UNIFY_VOID_γ $\Leftarrow$ unify_void(γ)

The unify_x_variable specializations:

- UNIFY_VARIABLE_Xα $\Leftarrow$ unify_x_variable(α)

The unify_x_value specializations:

- UNIFY_VALUE_Xα $\Leftarrow$ unify_x_value(α)

The unify_x_local_value specializations:

- UNIFY_LOCAL_VALUE_Xα $\Leftarrow$ unify_x_local_value(α)

The unify_y_variable specializations:

- UNIFY_VARIABLE_Yα $\Leftarrow$ unify_y_variable(α)

The unify_y_value specializations:

- UNIFY_VALUE_Yα $\Leftarrow$ unify_y_value(α)

The unify_y_local_value specializations:

- UNIFY_LOCAL_VALUE_Yα $\Leftarrow$ unify_y_local_value(α)

The previously introduced Quintus combination allocate_get_y_variable_x0 specialized into:

- ALLOCATE_GET_Yα_VARIABLE_X0 $\Leftarrow$ allocate, get_y_variable(α, 0)

The previously reintroduced combination unify_y_first_variable specialized into:

- UNIFY_Yα_FIRST_VARIABLE $\Leftarrow$ allocate, unify_y_variable(α, 0)

The progress opcode, (deallocate, execute(true)) as well as combinations of it with a preceding cut_y:

- PROGRESS $\Leftarrow$ deallocate, execute(true/0)
- CUT_Y_PROGRESS(A) $\Leftarrow$ cut_y(A), deallocate, execute(true/0)

Allocate is combined with a specialized `get_y_variable` for the X0 register and with unify_y_variable to form

- `ALLOCATE_GET_Y`α`_VARIABLE_X0` $\Leftarrow$ `allocate`, `get_y_variable`$(\alpha, 0)$
- `UNIFY_Y_FIRST_VARIABLE(X)` $\Leftarrow$ `allocate`, `unify_y_variable(X)`

A common sequence of opcodes are a number of put_value opcodes followed by a deallocate followed by an execute. This has been exploited in the depart opcodes.

- `DEPART(P)` $\Leftarrow$ `deallocate`, `execute(P)`
- `DEPART_0(P)` $\Leftarrow$), `put_y_value`$(0, 0)$, `deallocate`, `execute(P)`
- `DEPART_1(P)` $\Leftarrow$ `put_y_value`$(1, 1)$, `put_y_value`$(0, 0)$, `deallocate`, `execute(P)`
- `DEPART_2(P)` $\Leftarrow$ `put_y_value`$(2, 2)$, `put_y_value`$(1, 1)$, `put_y_value`$(0, 0)$, `deallocate`, `execute(P)`
- `DEPART_3(P)` $\Leftarrow$ `put_y_value`$(3, 3)$, `put_y_value`$(2, 2)$, `put_y_value`$(1, 1)$, `put_y_value`$(0, 0)$,

`deallocate`, `execute(P)`

Combinations of two `unify_x_var` opcodes, specialized for all possible combinations of the four lowest register pairs as well as the general case of any two registers follows here. The general case `u2_xvar_xvar` reintroduced. Two equivalent `unify_x_var` after each other is a nop.

- `UNIFY_VARS_X`α_2`_X`β_2 $\Leftarrow$ `unify_x_variable`(α_2), `unify_x_variable`(β_2)
- `U2_XVAR_XVAR(X, Y)` $\Leftarrow$ `unify_x_variable(X)`, `unify_x_variable(Y)`

Put_variable specialized to specific argument registers (A), and specific registers with permanent variables (Y). General case put_y_variable.

- `PUT_A`α`_VARIABLE_Y`β $\Leftarrow$ `put_y_variable`(α, β)
- `PUT_A`α`_VARIABLE_YN(Y)` $\Leftarrow$ `put_y_variable`$(\alpha$, `Y`$)$
- `PUT_AN_VARIABLE_Y`α`(X)` $\Leftarrow$ `put_y_variable`$($`X`$, \alpha)$

The specialization of `put_x_void`.

- `PUT_A`α`_VOID` $\Leftarrow$ `put_void`(α)

The specialization of `get_x_variable`.

- `GET_A`α_2`_VARIABLE_X`β_2 $\Leftarrow$ `get_x_variable`(α_2, β_2)
- `GET_A`α`_VARIABLE_XN(X)` $\Leftarrow$ `get_x_variable`$(\alpha$, `X`$)$
- `GET_AN_VARIABLE_X`β`(A)` $\Leftarrow$ `get_x_variable`$($`A`$, \beta)$

The specializations of `put_y_value(A, X)`:

- `PUT_A`α`_VALUE_Y`β $\Leftarrow$ `put_y_value`(α, β)
- `PUT_A`α`_VALUE_YN(X)` $\Leftarrow$ `put_y_value`$(\alpha$, `X`$)$

- PUT_AN_VALUE_Yβ(A) $\Leftarrow$ put_y_value(A, α)

The specializations of put_y_unsafe_value:

- PUT_Aα_UNSAFE_VALUE_Yβ $\Leftarrow$ put_y_value(α, β)
- PUT_Aα_UNSAFE_VALUE_YN(Y) $\Leftarrow$ put_y_value(α, Y)
- PUT_AN_UNSAFE_VALUE_Yβ(A) $\Leftarrow$ put_y_value(A, α)

The specialization of get_y_variable:

- GET_Aα_VARIABLE_Yβ $\Leftarrow$ get_y_variable(α, β)
- GET_Aα_VARIABLE_YN(Y) $\Leftarrow$ get_y_variable(α, Y)
- GET_AN_VARIABLE_Yβ(X) $\Leftarrow$ get_y_variable(X, β)

The specialization of get_x_value:

- GET_Aα_2_VALUE_Xβ_2 $\Leftarrow$ get_x_value(α_2, β_2)
- GET_Aα_VALUE_XN(X) $\Leftarrow$ get_x_value(α, X)
- GET_AN_VALUE_Xα(A) $\Leftarrow$ get_x_value(A, α)

M_3's 244 opcodes

The M_3 machine contains a subset of the improvements introduced in M_2, but it is an extension of M_1 instead of M_0.

The put_x_value opcode was translated into the get_x_variable and the put_xval_xval was translated into get_xvar_xvar. Then the put_x_value opcode was removed. The put_xval_xval could also have been removed since it is redundant. Five of the specializations introduced in M_2 were also implemented on top of M_1's optimizations to form M_3.

The specialization of unify_x_variable.

- UNIFY_VARIABLE_Xα $\Leftarrow$ unify_x_variable(α)

The specialization of (get_x_variable).

- GET_Aα_2_VARIABLE_Xβ_2 $\Leftarrow$ get_x_variable(α_2, β_2)
- GET_Aα_VARIABLE_XN(X) $\Leftarrow$ get_x_variable(α, X)
- GET_AN_VARIABLE_Xβ(A) $\Leftarrow$ get_x_variable(A, β)

The specialization of get_list.

- GET_Aα_LIST $\Leftarrow$ get_list(α)

The specialization of get_structure(A, A_2).

- GET_Aα_STRUCTURE(A) $\Leftarrow$ get_structure(A, α)

The specialization of the combination $\texttt{unify\_vars}(\texttt{X}, \texttt{Y})$ that exists in M_1.

- $\texttt{UNIFY\_VARS\_X}\alpha_2\texttt{\_X}\beta_2 \Leftarrow \texttt{unify\_x\_variable}(\alpha_2), \texttt{unify\_x\_variable}(\beta_2)$
- $\texttt{UNIFY\_VARS\_X}\alpha\texttt{\_XN}(\texttt{X}) \Leftarrow \texttt{unify\_x\_variable}(\alpha), \texttt{unify\_x\_variable}(\texttt{X})$
- $\texttt{UNIFY\_VARS\_XN\_X}\alpha(\texttt{X}) \Leftarrow \texttt{unify\_x\_variable}(\texttt{X}), \texttt{unify\_x\_variable}(\alpha)$