Empirical Parallel Performance Prediction from Semantics-based Profiling

Norman Scaife[1], Greg Michaelson[2], and Susumu Horiguchi[3]

[1] VERIMAG, Centre Equation, 2, Ave de Vignat, 38610 Giers, France.
`Norman.Scaife@imag.fr`

[2] School of Mathematical and Computer Sciences. Heriot-Watt University, Edinburgh, Scotland, EH14 4AS. `greg@cee.hw.ac.uk`

[3] Department of Computer Science, Graduate School of Information Sciences, Tohoku University, Aobayama 6-3-09, Sendai 980-8579, Japan.
`susumu@ecei.tohoku.ac.jp`

Abstract. The PMLS parallelizing compiler for Standard ML is based upon the automatic instantiation of algorithmic skeletons at sites of higher order function use. PMLS seeks to optimise run-time parallel behaviour by combining skeleton cost models with Structural Operational Semantics rule counts for HOF argument functions. In this paper, the formulation of a general rule count cost model as a set of over-determined linear equations is discussed, and their solution by singular value decomposition, and by a genetic algorithm, are presented.

1 Introduction

The PMLS (Parallelising ML with Skeletons) compiler for Standard ML [9] translates instances of a small set of common higher-order functions (HOFs) into parallel implementations of *algorithmic skeletons*. As part of the design of the compiler, we wish to implement performance-improving transformations guided by dynamic profiling. We contend that the rules that form the dynamic semantics of Standard ML provide an ideal set of counting points for dynamic profiling since they capture the *essence* of the computation at an appropriate level of detail. They also arise naturally during the evaluation of an SML program, eliminating difficult decisions about where to place counting points. Finally, the semantics provides an implementation independent basis for counting.

Our approach follows work by Bratvold [4] who used SOS rule counting, plus a number of other costs, to obtain sequential performance predictions for unnested HOFs. Bratvold's work built on Busvine's sequential SML to Occam translator for linear recursion [5] and was able to relate abstract costs in the SML prototype to specific physical costs in the Occam implementation.

Contemporaneous with PMLS, the FAN framework[2] uses costs to optimise skeleton use through transformation. FAN has been implemented within META[1] and applied to Skel-BSP, using BSP cost models and parameterisations. However, costs of argument functions are not derived automatically.

Alt *et al.*[3] have explored the allocation of resources to Java skeletons in computational Grids. Their skeleton cost models are instantiated by counting instruction executions in argument function byte code and applying an instruction timing model for the target architecture. As in PMLS, they solve linear equations of instruction counts from sequential test program runs to establish the timing model. However, the approach does not seem to have been realised within a compiler.

Hammond *et al.*[7] have used Template Haskell to automatically select skeleton implementations using static cost models at compile time. This approach requires substantial programmer involvement, and argument function costs are not derived automatically.

The main goal of our work is to provide predictions of sequential SML execution times to drive a transformation system for an automated parallelizing compiler. In principle, purely static methods may be used to derive accurate predictions, but for very restricted classes of program. From the start, we wished to parallelise arbitrary SML programs and necessarily accepted the limitations of dynamic instrumentation, in particular incomplete coverage and bias in test cases leading to instability and inaccuracy in predictions. However, we do not require predictions to be highly accurate so long they order transformation choices correctly.

In the following sections, we present our method for statistical prediction of SML based on the formal language definition, along with a set of test programs. We discuss the accuracy of our method and illustrate its potential use through a simple example program.

2 Semantic rules and performance prediction

The SML definition[10] is based on Structural Operational Semantics (SOS) where the evaluation of a language construct is defined in terms of the evaluation of its constituent constructs. Our methodology for dynamic profiling is to set up a dependency between rule counts and program execution times, and solve this system on a learning-set of programs designated as "typical".

Suppose there are N rules in an SOS and we have a set of M programs. Suppose that the time for the ith program on a target architecture is T_i, and that the count for the jth rule when the ith program is run on a sequential SOS-based interpreter is R_{ij}. Then we wish to find weights W_j to solve the M equations:

$$R_{i1}W_1 + R_{i2}W_2 ... + R_{iN}W_N = T_i$$

This linear algebraic system can be expressed in matrix form as:

$$RW \; = \; T \tag{1}$$

Then given a set of rule counts for a new program P we can calculate a good prediction of the time on the target architecture T_P from:

$$R_{P1}W_1 + R_{P2}W_2 ... + R_{PN}W_N = T_P \tag{2}$$

These are then substituted into skeleton cost models. For the currently supported list HOFs `map` and `fold`, the models take the very simple form:

$$par\ cost\ =\ C_1 * list\ size + C_2 * send\ size + C_3 * receive\ size + C_4 * arg\ \ cost \quad (3)$$

The coefficients $C_1 ... C_4$ are determined by measurements on the target architecture, over a restricted range of a set of likely parameters[12]. We then deploy a similar fitting method to this data, relating values such as communications sizes and instance function execution times to measured run-times.

3 Solving and predicting

We have tried to generate a set of test programs which, when profiled, include all of the rules in the operational semantics which are fired when our application is executed. We have also tried to ensure that these rules are as closely balanced as possible so as not to bias the fit towards more frequently-used rules.

We have divided our programs into a *learning* and a *test* set. The learning set consists of 99 "known" programs which cover a fair proportion of the SML language. These include functions such as mergesort, maximum segment sum, regular expression processing, random number generation, program transformation, ellipse fitting and singular value decomposition.

The test set consists of 14 "unknown" programs which, in turn, represent a fair cross-section of the learning set in terms of the sets of rules fired and the range of execution times. These include polynomial root finding, least-squares fitting, function minimisation and geometric computations. The test set was generated by classifying the entire set of programs according to type (e.g. integer-intensive computation, high-degree of recursion) and execution time. A test program was then selected randomly from each class.

To generate the design matrix R, we take the rule counts R_i^{td} and execution time T_i^{td} for top level declaration number td. The first timing T_i^0 in each repeat sequence is always ignored reducing the effect of cache-filling. The execution times $T_i^{t_i}$ are always in order of increasing number of repeats such that $T_i^x < T_i^y$ for $x < y$. Using this and knowing that outliers are always greater than normal data we remove non-monotonically increasing times within a single execution. Thus if $T_i^{td-1} < T_i^{td} < T_i^{td+1}$ then the row containing T_i^{td} is retained in the design matrix. Also, to complete the design matrix, rules in R_{all} which are not in R_i^{td} are added and set to zero.

Some rules can be trivially removed from the rule set such as those for type checking and nesting of expressions with atomic expressions. These comprise all the rules in the static semantics. However, non-significant rules are also removed by totaling up the rule counts across the entire matrix. Thus for rule r_x and a threshold θ, if:

$$\sum_{i=0}^{n} \sum_{j=0}^{t_i} R_i^j[r_x].c < \theta \sum_{i=0}^{n} \sum_{j=0}^{t_i} R_i^j[r_{max}].c \quad (4)$$

r_{max} is the most frequent rule and $R_i^j[r_k].c$ means the count for rule r_k in the list of rule counts R_i^j. Thus rules with total counts less than a threshold value times the most frequently fired rule's total count have their columns deleted from the rule matrix R. This threshold is currently determined by trial and error. The execution time vector T_n is generated from the matching execution times for the surviving rows in the rule matrix.

Fitting is then performed and the compiler's internal weights updated to include the new weights. Performance prediction is then a simple application of Equation 1, where R is the set of rules remaining after data-workup and W is the set of weights determined by fitting. For verification, the new weights are applied to the original rule counts giving reconstructed times T_{recon} and are compared with the original execution times T_n.

Once the design matrix is established using the learning set, and validated using the test set, we can then perform fitting and generate a set of weights. We have experimented with *singular value decomposition* (SVD) to solve the system as a linear least-squares problem[11]. We have also adapted one of the example programs for our compiler, a parallel *genetic algorithm* (GA) [8], to estimate the parameters for the system.

4 Accuracy of fitting

Our compilation scheme involves translating the Standard ML core language as provided by the ML Kit Version 1 into Objective Caml, which is then compiled (incorporating our runtime C code) to target the parallel architecture. We have modified the ML Kit, which is based closely on the SML SOS, to gather rule counts directly from the sequential execution of programs.

Using an IBM RS/6000 SP2, we ran the 99 program fragments from the learning set using a modest number of repeats (from 10 to about 80, depending upon the individual execution time). After data cleanup, the resulting design matrix covered 41 apply functions[1] and 36 rules from the definition, and contained 467 individual execution times. Applying the derived weights to the original fit data gives the levels of accuracy over the 467 measured times shown in Figure 1. This table presents a comparison of the minimum, maximum, mean and standard deviation of the measured and reconstructed times for both fitting methods. The same summary is applied to the percentage error between the measured and reconstructed times.

First of all, the errors computed for both the learning and test sets look very large. However, an average error of 25.5% for SVD on the learning set is quite good considering we are estimating runtimes which span a scale factor of about 10^4. Furthermore, we are only looking for a *rough* approximation to the absolute values. When we apply these predictions in our compiler it is often the *relative* values which are more important and these are much more accurate although more difficult to quantify.

[1] Apply functions are external primitive functions called by the SML core language.

	Fit	χ^2	Time (s)	Min	Max	Mean	Std. Dev.
Learning	x		Measured	5.11×10^{-6}	0.00235	0.000242	0.000425
Set	SVD	4.1×10^{-7}	Reconstructed	-2.65×10^{-6}	0.00239	0.000242	0.000424
		% Error	0.00571%	267.0%	25.5%	41.3%	
	GA	4.9×10^{-5}	Reconstructed	5.98×10^{-8}	0.00163	0.000179	0.000247
		% Error	0.00977%	1580.0%	143.0%	249.0%	
Test Set	x		Measured	8.61×10^{-6}	0.0399	0.00221	0.0076
	SVD		Reconstructed	-8.06×10^{-5}	0.0344	0.00195	0.00656
		% Error	0.756%	836.0%	158.0%	208.0%	
	GA		Reconstructed	1.67×10^{-7}	0.01600	0.000965	0.000304
		% Error	1.56%	284.0%	67.9%	71.1%	

Fig. 1. Summary of fit and prediction accuracy

The SVD is a much more accurate fit than GA as indicated by the χ^2 value for the fit. However, the SVD fit is much less stable than the GA fit as evidenced by the presence of negative reconstructed times for SVD. This occurs at the very smallest estimates of runtime near the boundaries of the ranges for which our computed weights are accurate. The instability rapidly increases as the data moves out of this region.

5 Performance Prediction Example

As part of the PMLS project we have used *proof-planning* to construct a synthesiser which extracts HOFs from arbitrary recursive functions[6]. For example, given the following program which squares the elements of a list of lists of integers:

```
fun squares [] = []
  | squares ((h:int)::t) = h*h::squares t

fun squs2d [] = []
  | squs2d (h::t) = squares h::squs2d t
```

the synthesizer generates the six programs shown in Figure 2. Note that there is no parallelism in this program suitable for our compiler and we would expect our predictions to validate this.

We require the execution times for the instance functions to the `map` and `foldr` HOFs. We have not yet automated the collection of this data or linked the output from the performance prediction into the compiler so we present a hand analysis of this code.

Figure 3 shows the predicted instance function execution times for the two fitting methods alongside the actual measured times. The input data is a 5×5 list of lists of integers. The predictions are in roughly the correct range but differ significantly from the measured times. Despite the greater accuracy of the SVD fit to the learning-set data, the GA-generated weights give more consistent results compared to actual measured values. This is due to the numerical

```
1. val squs2d = fn x => map (fn y => map (fn (z:int) => z*z) y) x
2. val squs2d =
      fn x => foldr (fn y => fn z => (map (fn (u:int) => u*u) y::z)) [] x
3. val squs2d =
      fn x => map (fn y => foldr (fn (z:int) => fn u => z*z::u) [] y) x
4. val squs2d =
       fn x => foldr (fn y => fn z =>
                      foldr (fn (u:int) => fn v => u*u::v) [] y::z) [] x
5. val squs2d = fn x => map (fn y => squares y) x
6. val squs2d = fn x => foldr (fn y => fn z => squares y::z) [] x
```

Fig. 2. Synthesizer output for `squs2d`

V	Position	HOF	Rules	T_{SVD}	T_{GA}	$T_{measured}$
1	outer	map	21	2.63	5.56	8.61
	inner	map	8	0.79	1.40	3.36
2	outer	fold	21	4.97	6.01	9.17
	inner	map	8	0.79	1.40	3.14
3	outer	map	20	1.73	7.53	12.6
	inner	fold	15	12.5	3.66	3.71
4	outer	fold	20	4.06	7.98	11.1
	inner	fold	15	12.5	3.66	3.53
5	single	map	19	3.58	3.45	6.65
6	single	fold	19	5.91	3.90	7.97

Fig. 3. Predicted and measured instance function times (μS)

instability of the SVD fit. However, these discrepancies are sufficient to invert the execution times for nested functions. For instance, for Version 3 the inner fold instance function takes longer than the outer one, even though the outer computation encompasses the inner.

Applying the skeleton performance models to the measured instance function times, plus data on communications sizes gathered from sequential executions, gives the predicted parallel run-times for 1, 2, 4 and 8 processors, shown in Figure 4.

The GA- and SVD-predicted instance function times give identical predictions for parallel run-times. This is because the parallel performance model is in a range where the run-time is dominated by communications rather than computation. However, the P_1 predictions are erroneous. These predictions represent an extrapolation of a parallel run onto a sequential one which has no overheads such as communication. This also applies to the P_2 predictions, where these overheads are not accurately apportioned. Furthermore, the absolute values of the predictions are unreliable. For the P_8 values, some are accurate but some are out by an order of magnitude. The most relevant data in this table is the ratio between the P_4 and P_8 values. This, in most cases, increases as the number of processors increases, indicating slowdown.

V	Position	HOF	P/M	P_1	P_2	P_4	P_8
1	outer	map	P	1.6000	3.230	6.480	12.990
			M	0.1423	6.806	5.279	4.910
	inner	map	P	3.2700	4.900	8.150	14.660
			M	0.2846	35.200	15.620	14.440
2	outer	fold	P	7.3700	10.940	18.070	32.340
			M	0.1617	4.204	3.101	3.634
	inner	map	P	3.2700	4.900	8.150	14.660
			M	0.3040	35.360	14.900	14.940
3	outer	map	P	1.6000	3.230	6.480	12.990
			M	0.2205	7.314	3.923	4.739
	inner	fold	P	14.2000	17.760	24.900	39.170
			M	0.3875	26.020	14.570	15.770
4	outer	fold	P	7.3700	10.940	18.070	32.340
			M	0.2344	5.058	2.907	4.047
	inner	fold	P	14.2000	17.760	24.900	39.170
			M	0.3907	23.080	13.200	16.110
5	single	map	P	1.6000	3.230	6.480	12.990
			M	0.1375	6.590	4.092	4.570
6	single	fold	P	7.3700	10.940	18.070	32.340
			M	0.1587	4.024	3.002	3.750

Fig. 4. Predicted (P) and measured (M) parallel run-times (mS)

6 Conclusions

Overall, our experimentation gives us confidence that combining automatic profiling with cost modeling is a promising approach to performance prediction. We now intend to use the system as it stands in implementing a performance-improving transformation system for a subset of the SML language. As well as exploring the automation of load balancing, this gives us a further practical way to assess the broader utility of our approach.

While we have demonstrated the feasibility of semantics-based profiling for an entire extant language, further research is needed to enable more accurate and consistent predictions of performance from profiles. Our work suggests a number of areas for further study.

It would be useful to identify which semantic rules counts are most significant for predicting run times, through standard statistical techniques for correlation and factor analyses. Focusing on significant rules would reduce profiling overheads and might enable greater stability in the linear equation solutions. Furthermore, non-linear costs might be introduced into the system, relating profile information and runtime measurements. The system would no longer be in matrix form and would require the use of generalised function minimisation instead of deterministic fitting. Predictions might also be made more accurate by investigating the effects of optimisations employed in the back end compiler, which fundamentally affect the nature of the association between the language semantics and implementation. Our studies to date have been of very simple functions and of unrelated substantial exemplars: it would be worth systematically exploring the relationship between profiles and run-times for one or more constrained classes of recursive constructs, in the presence of both regular and irregular computation patterns. Finally, aspects of implementation which are subsumed in the semantics notation might be modeled explicitly, in particular the creation and manipulation of name/value associations which are hidden behind the semantic notion of environment.

7 Acknowledgments

This work was supported by Postdoctoral Fellowship P00778 of the Japan Society for the Promotion of Science (JSPS) and by UK EPSRC grant GR/L42889.

References

1. M. Aldinucci. Automatic Program Transformation: The META Tool for Skeleton-based Languages. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, volume 10 of *Advances in Computation: Theory and Practice*. NOVA Science, 2002.
2. M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelegatti. Towards Parallel Programming by Transformation: The FAN Skeleton Framework. *Parallel Algorithms and Applications*, 16(2-3):87–122, March 2001.
3. M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.
4. T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes*. PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.
5. David Busvine. Implementing Recursive Functions as Processor Farms. *Parallel Computing*, 19:1141–1153, 1993.
6. A. Cook, A. Ireland, G. Michaelson, and N. Scaife. Deriving Applications of Higher-Order Functions through Proof Planning. *Formal Aspects of Computing*, accepted Nov. 2004.
7. K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003.
8. G. Michaelson and N.Scaife. Parallel functional island model genetic algorithms through nested skeletons. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on the Implementation of Functional Languages*, pages 307–313, Aachen, September 2000.
9. G. Michaelson and N. Scaife. Skeleton Realisations from Functional Prototypes. In F. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT, 1990.
11. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. CUP, 2nd edition, 1992.
12. N. R. Scaife. *A Dual Source, Parallel Architecture for Computer Vision*. PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1996.