Optimized Selection of Runtime Mode for the Reconfigurable PRAM-NUMA Architecture REPLICA Using Machine-Learning

Erik Hansson and Christoph Kessler

Dept. of Computer and Information Science
Linköpings universitet, Sweden
{eriha,chrke}@ida.liu.se

Abstract. The massively hardware multithreaded VLIW emulated shared memory (ESM) architecture REPLICA has a dynamically reconfigurable on-chip network that offers two execution modes: PRAM and NUMA. PRAM mode is mainly suitable for applications with high amount of thread level parallelism (TLP) while NUMA mode is mainly for accelerating execution of sequential programs or programs with low TLP. Also, some types of regular data parallel algorithms execute faster in NUMA mode. It is not obvious in which mode a given program region shows the best performance. In this study we focus on generic stencil-like computations exhibiting regular control flow and memory access pattern. We use two state-of-the art machine-learning methods, C5.0 (decision trees) and Eureqa Pro (symbolic regression) to select which mode to use.We use these methods to derive different predictors based on the same training data and compare their results. The accuracy of the best derived predictors are 95% and are generated by both C5.0 and Eureqa Pro, although the latter can in some cases be more sensitive to the training data. The average speedup gained due to mode switching ranges between 1.92 to 2.23 for all generated predictors on the evaluation test cases, and using a majority voting algorithm, based on the three best predictors, we can eliminate all misclassifications.

1 Introduction

In the multicore era we do not only face the problems that parallel programming brings; modern architectures and hardware platforms also expose the advantages and problems of managing heterogeneity. Today's computer systems usually have multicore processor chips and dedicated accelerators such as GPUs. To utilize these systems efficiently, boils down to selecting where and how to run a program. How to achieve high performance for real applications ist not straight forward, and predicting performance is even harder since aspects such as data locality and movement has to be considered.

In this study we use the VLIW massively hardware multithreaded emulated shared memory (ESM) architecture REPLICA. Each core has 512 hardware threads and the processor pipeline is designed so that the high number of threads

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 133–145, 2014.
© Springer International Publishing Switzerland 2014

effectively can hide the latency of accessing the emulated shared memory. Since it is realizes the PRAM (parallel random access machine) model [12] it is very convenient to program. To get full performance an ESM needs programs with large enough thread level parallelism (TLP). To solve the problem with low TLP REPLICA can be reconfigured at run time so that the time slot of several hardware threads are bunched together and access on-chip memory modules in NUMA mode such that the PRAM emulation is switched off and the overhead from plain ESM is removed [6,7]. Switching between PRAM and NUMA mode take only a moderate number of clock cycles as overhead.

For the programmer, NUMA mode means that there are fewer threads and the memory latency becomes "visible" and has to be taken care of manually to utilize the hardware fully. The main reason for having NUMA mode is to be able to accelerate execution of sequential legacy programs and programs with low thread level parallelism faster since they do not suit PRAM mode very well [8]. To switch to NUMA mode the programmer can join all the threads on a core at runtime, so each core becomes single threaded, and can execute faster. It is not always obvious which parallel programs will run faster in NUMA mode, one reason is that hashing of memory adresses is not exposed to the programmer. To tackle this we use state-of-the-art machine-learning methods.

We have in earlier work, for example in [8], introduced REPLICAs PRAM-NUMA programming model and given som basic examples and evaluations. We have also earlier done a preliminary evaluation of REPLICA PRAM capabilities [11], where one conclusion was that PRAM mode is very good for irregular memory access and control flow problems in contrast to commercially available state-of-the-art CPUs and GPUs. However, REPLICA PRAM mode was in several cases outperformed by cache based CPUs and GPUs when it comes to regular memory accesses and control computations [11].

The main goal of this paper is to define an initial model that predicts when to use NUMA mode and when to use PRAM mode in terms of performance.

Since PRAM mode already is very fast for irregular problems but possible suboptimal for regular [11], we here focus on regular data parallel problems namely, generic stencil computations. In [8] we also showed that locality and latency optimizations could be beneficial in NUMA mode, these optimizations suite of course regular problems well.

A secondary goal of this paper is to take two popular state-of-the-art machine learning tools, one based on decision trees and one on symbolic regression, to see if they can be used for modeling this kind of performance optimization problems of heterogeneous architectures. For both methods we use the same training and evaluation data sets. The accuracy of the best derived predictors are 95% and are generated by both C5.0 and Eureqa Pro, although the later can in some cases be more sensitive to the training data. The average speedup gained due to mode switching ranges between 1.92 to 2.23 for all generated predictors on the evaluation test cases. Using a majority voting algorithm, based on the three best predictors, we can eliminate all misclassifications on the evaluation test cases.

2 REPLICA Architecture

The REPLICA architecture is a *chip multiprocessor* (CMP) family of *configurable emulated shared memory machines* (CESM) [8] designed by VTT, Oulu, Finland[1]. Different configurations of the processor have different numbers of cores', arithmetical logical units (ALUs), and memory units (MUs). The processor cores memory modules are connected via a 2D multimesh network. In this study we use the cycle accurate REPLICA simulator, however a hardware prototype is under construction.

2.1 PRAM Mode

One main feature of the REPLICA architecture is PRAM mode. In PRAM mode, the programming model of the processor exposed to the programmer is the *Concurrent Read Concurrent Write* (CRCW) PRAM model. It gives a deterministic synchronous programming model that allows concurrent memory accesses and strict memory consistency [12]. To hide the memory latency each core has 512 hardware threads. Each REPLICA core is a VLIW architecture; in PRAM mode it supports chained functional units (FUs) which means that the result of one functional unit can be used as input to the next unit in the pipeline in the same step. This reduces the pressure on general purpose registers and we are not dependent on the same degree of instruction level parallelism (ILP) as ordinary VLIW architectures are to utilize all functional units. We have specific code generation support and an optimization phase in our LLVM based REPLICA compiler to support arbitrary numbers of chained functional units [13].

In PRAM mode REPLICA supports so called multiprefix instructions [5]. Threads in the same thread group that execute the same multiprefix instruction participate to calculate the result together. Multiprefix operations are considered important building blocks in parallel algorithms, see for example [12]. Programming PRAM mode is straight forward, but to get good performance out of PRAM mode the programmer should use the multiprefix operations if possible.

2.2 NUMA Mode

As mentioned before, the main motivation for NUMA mode is to be able to accelerate execution of sequential legacy programs and programs with low thread level parallelism faster than in PRAM mode. To switch to NUMA mode REPLICA has a specific assembly instruction `JOIN` that joins all threads in a thread group to a NUMA bunch. To switch back to PRAM mode we use the `SPLIT` instruction. In our C based REPLICA baseline language [8] we have a construct **numa**(s) that switches the processor to NUMA mode, executes the statement s, and switches back to PRAM mode. The construct also orchestrates setting up the stack pointers, thread ids etc. and restores them. The overhead of switching to NUMA and

[1] REPLICA project site: `http://www.vtt.fi/sites/replica/?lang=en`

back is around 16000 clock cycles[2] [8]. When the processor runs in NUMA mode, no chaining of functional units is possible, the number of functional units is fixed to the ones given in Table 1 even though it can have more in PRAM mode. This is taken care of by the compiler [8]. Programming for NUMA mode is like programming for traditional NUMA multicore processor with global shared memory and private local memory.

In this paper we have selected a configuration (number of ALUs and MUs) in PRAM mode that looks most similar to the fixed one in NUMA mode, to be able to highlight the differences that comes from NUMA mode itself and not from having a fat PRAM. See Table 1 for the specific processor configurations.

Table 1. Configurations used in this paper

Mode	Cores	Threads per core	Pre memory ALU	MU	Post Memory ALU	Compare unit	Chained FU
PRAM	4	512	1	1	0	1	Yes
NUMA	4	1	1	1	0	1	No

There are three paradigms for accessing shared memory in NUMA mode [8]:
- Freeze processor: the whole processor freezes until the data has arrived.
- Freeze bunch: only the bunch freezes until data has arrived.
- Load with explicit receive (LER): After an asynchronous shared load, an explicit receive instruction `RECO` need to be issued; in between, other instructions can execute.

These paradigms are fixed and can not be selected at runtime. If a `RECO` is issued directly after a load the bunch will freeze until data has arrived just like int the case of "freeze bunch". It important to note that loding from shared memory with the LER paradigm occupies the memory unit twice, once for the load instruction (`LDO`) and once for the receive (`RECO`). With LER the result of the load will be stored in a register and used as an input to the receive. The used register will be kept alive until the receive is done. If we are short of the registers the compiler will insert spill code to local memory (stack) wich can reduce performance and it might have ben better to "freeze" the bunch instead. In this paper we still only focus on LER, since it gives more opportunities for optimizations.

3 Parameterized Benchmark

In earlier initial work we have shown that PRAM mode suits programs with irregular memory access and control flow very well, while regular problems do not benefit from PRAM mode [11]. In that study we only focused on PRAM

[2] About 11000 cycles for PRAM-NUMA switching and about 5000 vice versa. As there are 512 PRAM threads per core in PRAM mode, this corresponds to about 32 PRAM instructions executing per thread for switching back and forth again.

mode, still our experience is that in order to be able to make (any) practical use of NUMA the problems needs to be very regular in terms of memory accesses and control flow. We need to reduce loads and stores from shared memory, but also reading and writing to local memory is considered expensive in NUMA mode.

To explore when NUMA mode can be beneficial we introduce a parameterized benchmark that is very regular. It can be considered a stencil operation. Compared to other regular algorithms, such as vector and matrix operations, stencil operations are more generic especially regarding how much computation there is per data element. We apply register pipelining [4] in order to load each element once (recall that REPLICA has no caches), and obtain a typical software pipeline code structure consisting of prologue (filling the pipeline), computational kernel (steady state) and epilogue (draining the pipeline). We have the following parameters:

- N: Problem size (number of array elements to update)
- P: Prologue size
- C: Number of instructions for local computations with no memory access
- LLS: Number of local loads and stores.

P models how many times we run the prologue, eg. the prefetching of data in shared memory that we have to do before the kernel can start execute. The computational kernel loop run for N iterations, once for each data element. Inside the kernel we do LLS local loads and stores. Our experience shows that in order to be able to get any performance out of NUMA mode we can only access a shared memory data element once, otherwise the performance is ruined. If we need the same data again, we must keep it locally (in local memory but registers are preferred). The C parameter resembles the distance between the load and the receive (`RECO` instruction), and can be seen as how much local (register based) computation we need to do, to hide the latency of the shared memory access.

Optimizations done for NUMA, such as register pipelining [4] to avoid memory accesses (both shared and local) are in our experience often also useful in PRAM mode. To make a fair comparison between NUMA and PRAM mode we run the exactly same program with the same optimizations (locality, registers etc) for both NUMA and PRAM. The only difference is that we, of course, switch to NUMA and the back-end compiler compiles NUMA code to fit the NUMA pipeline (no chained FU). In NUMA mode we also access shared memory using the LER concept (loads with explicit receive instruction). In PRAM mode we also divide the work over all the available HW threads, e.g. 4×512. In NUMA mode we have joined all the threads per core, so we only have 4 cores (threads) to divide the work among.

4 Machine-Learning Models

Using a random number generator (with lower and upper limits on each parameter) we generated different training set and evaluation sets. The parameter space is well covered, see the distribution of the parameters for the evaluation set S_E in

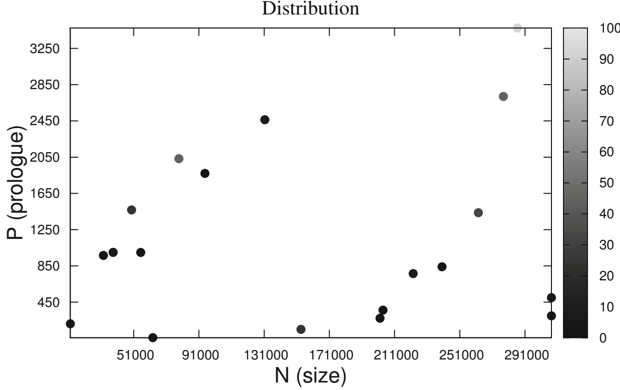

Fig. 1. Distribution of N (size), P (prologue) and LLS (local load store) parameters in the evaluation set S_E. LLS is shown in grayscale.

Figure 1. We generated programs in REPLICA baseline language (an extended version of C), compiled them and run them on the cycle accurate REPLICA simulator.

Initially we started with two training sets, S_1 and S_2 and their union S_3, where $|S_3| = 45$ and $|S_1| \approx |S_2|$. S_1 and S_2 are disjoint.

Using S_3 we derive formulas for both NUMA and PRAM execution times using the Eureqa Pro framework [18,17]. We assume that the execution time, t_n, for NUMA is $t_n = f_n(N, C, P, LLS)$ and t_p for PRAM in a similar way. Since we want to make a predictor for when to switch to NUMA mode it is natural to use the speedup, eg. only switch to NUMA if we get speedup larger than one, e.g. $\frac{t_p}{t_n} > 1$.

For C5.0 [16] we tried S_1, S_2 and S_3 as training sets, however the results were not good enough so we decided to increase the size of the training data. The new sets S_1', S_2' and $S_3' = S_1' \cup S_2'$ contains the old sets in the following way: $S_1 \subset S_1'$, $S_2 \subset S_2'$ where $S_3 \subset S_3'$, $|S_3'| = 98$ and $|S_1'| \approx |S_2'|$

For evaluation we use the set S_E, it is shown in Table 2. It is independent from the training sets and $|S_E| = 20$ [3]. Figure 1 depicts S_Es distribution of the parameters N (size), P (prologue) and LLS (local loads and stores) where LSS is shown with grayscale. Since C5.0 did not use the C parameter we do not show its distribution in the picture.

4.1 Eureqa Pro

A First Model Using S_3. With S_3 we got the following execution time models.

NUMA: $t_n = a_n * P + b_n * N + c_n * N * C + d_n * LLS^2$, where $a_n = 0$, $b_n = 9.65313889909994$, $c_n = 0.224715696133425$ and $d_n = 23005.0142108128$.

[3] Due to large simulation times, a significantly larger number of training and evaluation samples was not feasible.

PRAM: $t_p = a_p*P + b_p*N + c_p*N*C + d_p*LLS^2$ where $a_p = 3561.80442755642$, $b_p = 5.91351736227114$, $c_p = 0.232529731521942$ and $d_p = 8662.82648471137$.

As stated earlier we use the estimated speedup larger than one, $\frac{t_p}{t_n} > 1$, as a predictor. The generated expressions for the execution time, are quite similar for PRAM and NUMA, one interesting detail is that the P (prologue) is not used for NUMA and that $c_n \approx c_p$.

An Updated Model Using S'_3. When we double the training set size, adding more training cases, using S'_3, we get different formulas for estimating the PRAM and NUMA execution times.

NUMA: $t_n = a_n * LLS + b_n * N + c_n * N * LLS * \sqrt{N} + d_n * S * LLS^2$ where $a_n = 44201.802640319$, $b_n = 8.01445073789093$, $c_n = 0.0765782960682957$ and $d_n = -0.000128963606572698$.

PRAM: $t_p = a_p*LLS + b_p*P + N*LLS + c_p*P*LLS + d_p*N*C + e_p*LLS^2$ where $a_P = 107691.057352967$, $b_p = 2919.18227717314$, $c_p = 107.696490731206$, $d_p = 0.26706472514021$ and $e_p = -1605.81303793423$. Note that the coefficient for $S * LLS$ is 1.

Binary Model Using S'_3. Deriving the execution times as functions of N, P, C and LLS seems unstable for Eureqa Pro, i.e. very depending on the specific training sets (see Table 4). However, we are only interested in relative performance. To handle this we introduced a binary model using Eureqa Pro and S'_3. It gives 1 as result if NUMA should be used and 0 for PRAM: $NUMA = P > 0.000905177867360653 * N + 0.055818680067241 * P * LLS$.

4.2 C5.0 Decision Trees

Running C5.0 on S'_1 we get the following decision tree:

```
LLS > 8: PRAM
LLS<= 8:
:...P > 400: NUMA
    P <= 400:
    :...N <= 174851: NUMA
        N > 174851: PRAM
```

Running C5.0 on S'_2 we get the following decision tree:

```
LLS > 15: PRAM
LLS <= 15:
:...N <= 177417: NUMA
    N > 177417:
    :...P <= 643: PRAM
        P > 643: NUMA
```

Running C5.0 on S'_3 we get the following decision tree:

```
LLS > 8: PRAM
LLS <= 8:
:...N <= 210765: NUMA
    N > 210765:
    :...P <= 500: PRAM
        P > 500: NUMA
```

Note that no C5.0 model uses the C parameter and all the generated predictors are quite similar to each other.

4.3 Evaluation and Comparison of Eureqa Pro and C5.0 Models

Table 3 shows the results of evaluating the Eureqa Pro models based on S_3 and S_3' compared to actual results from test runs in the simulator. It might be interesting to note that the average speedup error is 15.1% for S_3 while it is 37.5% for the model based on S_3'. Table 4 shows the predicted mode for all cases in S_E for the models based on S_1', S_2',S_3' and S_3. The columns are sorted by the number of misclassifications and also show the average real speedup gained when using the corresponding predictors.

Table 2. The set of test cases, S_E, for evaluation and comparison

Case	N	C	P	LLS
1	307200	104	300	4
2	153600	12	150	24
3	307300	16	500	8
4	38400	12	1000	8
5	38400	24	1000	16
6	222401	16	765	8
7	62709	72	59	1
8	286382	16	3475	92
9	78714	56	2034	44
10	32431	88	966	4

Case	N	C	P	LLS
11	202099	56	273	4
12	240096	56	841	4
13	131407	40	2463	8
14	277705	96	2720	44
15	55311	56	999	1
16	12155	88	212	8
17	203909	48	362	1
18	262394	80	1437	32
19	49693	80	1467	24
20	94664	0	1871	1

For our problem and training sets Eureqa Pro using our first execution time based models seems to be some what unstable; it produces both their best and worst predictor depending on the training set, see Table 4. Eureqa Pro gives the best predictor with a smaller training set S_3 than the larger S_3'. With our binary Eureqa Pro predictor based on set S_3' we get only one misclassification, it seems as good as C5.0 for the same set. All predictors are very simple and can be implemented with a few instructions so the overhead of invoking them at runtime is very low.

All predictors give on average a speedup between 1.91 and 2.23. This is not a "magical range", it comes from the evaluation set S_E. In most cases misclassifications do not "hurt" so much since they are border cases and running in "wrong" mode will only affect performance marginally. If we remove the three worst predictors in terms of misclassifications and use the remaining predictors together with a majority voting algorithm we would eliminate all misclassifications in our evaluation set and get an average speedup of 2.23. In Table 4 the speedup for C5.0 on S_3' is also 2.23, even though it makes one misclassification, as this misclassification only changes the average speedup with 0.03% which can be explained by that there the PRAM and NUMA execution times are very similar. We also evaluated our different models on two different parallel 1D-average computations, the parameters and results are shown in Table 5. The only predictor that misclassifies is Eureqa on S_3'.

Table 3. Real speedup from simulator, estimated speedup for Eureqa Pro using S_3' and S_3 and the speedup error in % compared to the real speedup

Case	Real speedup	Eureqa S_3 speedup	Eureqa speedup error % S_3	Eureqa S_3' speedup	Eureqa speedup error % S_3'
1	0.98	0.99	1.5	1.83	86.6
2	0.42	0.45	7.3	0.17	60.4
3	0.98	0.96	3.0	0.66	33.2
4	5.45	2.29	58.1	1.33	75.7
5	0.84	0.96	14.7	0.95	13.0
6	1.26	1.23	2.3	0.42	66.9
7	0.99	1.00	1.2	1.25	26.8
8	0.41	0.45	7.7	0.61	46.4
9	0.56	0.55	2.2	0.52	7.3
10	3.79	3.35	11.5	2.69	28.9
11	1.02	1.02	0.1	0.60	40.8
12	1.31	1.35	2.5	0.93	29.5
13	3.03	2.89	4.6	0.98	67.6
14	0.47	0.64	37.8	0.70	49.7
15	3.92	3.68	6.1	3.68	6.1
16	2.04	0.89	56.3	1.74	14.8
17	1.19	1.14	4.5	1.19	0.0
18	0.46	0.66	45.1	0.51	12.1
19	0.77	0.78	2.1	0.85	10.5
20	11.60	7.72	33.5	3.12	73.1

Table 4. Evaluation result using S_E. Misclassifications are marked in boldface. Average speedup for all 20 cases using the corresponding predictor.

Case	Correct	Eureqa S_3'	C5.0 S_2'	C5.0 S_1'	Eureqa S_3	C5.0 S_3'	Eureqa bin S_3'
1	PRAM	**NUMA**	PRAM	PRAM	PRAM	PRAM	PRAM
2	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
3	PRAM	PRAM	**NUMA**	PRAM	PRAM	PRAM	PRAM
4	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
5	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	**NUMA**
6	NUMA	**PRAM**	NUMA	NUMA	NUMA	NUMA	NUMA
7	PRAM	**NUMA**	**NUMA**	**NUMA**	PRAM	**NUMA**	PRAM
8	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
9	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
10	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
11	NUMA	NUMA	**PRAM**	**PRAM**	NUMA	NUMA	NUMA
12	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
13	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
14	PRAM	**NUMA**	PRAM	PRAM	PRAM	PRAM	PRAM
15	NUMA	**PRAM**	NUMA	NUMA	NUMA	NUMA	NUMA
16	NUMA	**PRAM**	NUMA	NUMA	**PRAM**	NUMA	NUMA
17	NUMA	NUMA	**PRAM**	**PRAM**	NUMA	NUMA	NUMA
18	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
19	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
20	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
	Misclassifications	**6**/20	**4**/20	**3**/20	**1**/20	**1**/20	**1**/20
	Average real speedup	1.92	2.21	2.21	2.01	2.23	2.22

Table 5. Parameters and evaluation results using a 1D average computation example. Misclassifications are marked in boldface.

Evaluation	N	C	P	LLS	Correct	Eureqa S_3'	C5.0 S_2'	C5.0 S_1'	Eureqa S_3	C5.0 S_3'	Eureqa bin S_3'
1	307200	60	8	1	PRAM	**NUMA**	PRAM	PRAM	PRAM	PRAM	PRAM
2	307200	80	8	1	PRAM	**NUMA**	PRAM	PRAM	PRAM	PRAM	PRAM

5 Related Work

The problem of selecting the best runtime mode for REPLICA is very related to the implementation selection problem for heterogenous systems such as CPU-GPU based systems; in both REPLICA and CPU-GPU case there are overhead costs of switching and data transfers costs. Similarities such as small local memory also exist.

The parallel programming language framework PetaBricks [1] uses auto-tuning that effectively explores the search space to select from multiple user provided implementations the best one, depending on problem parameters [1]. Their compiler can generate OpenCL code that can execute on GPUs [15].

SkePU is a C++ skeleton programming library mainly for mapreduce problems with back-ends for both CPU and GPUS. It supports implementation selection using machine-learning methods to adopt skeletons to a given platform [3].

One example where C5.0 has been used for performance optimization of heterogeneous systems is [14], they use it to prune the search space when doing off-line tuning of component composition.

Danylenko et al. compared different machine-learning approches for context-aware composition in [2]. They consider decision tress, decision diagrams, naive bayes and suport vector machine classifiers. They evaluate their results on three different multicore machines.

In [10] Grewe and O'Boyle show how to select optimized mappings of OpenCL tasks on a heterogeneous CPU-GPU system to get good load balancing. They base their training on the support vector machine (SVM) model, and is based on static features (number of floating point operations etc) just like we do. However hybrid execution is not possible on REPLICA as the same hardware is used by both PRAM and NUMA mode.

Elastic computing is a framework that supports heterogenous computing, such multicore CPUs combined with FPGA accelerators. It separates functionality and implementations using elastic functions which can be executed with different parameters (input size etc) on different target architectures [20]. They use linear regression based to predict execution time based input size and other metrics captured by the specific performance model for each component.

As far as we know, Eureqa Pro has not been used before for performance prediction and implementation selection before, however in [9] Goel uses Eureqa for per-core power estimation and power aware scheduling for CMPs which is a related problem area.

6 Conclusion and Future Work

We used state-state-of-the-art machine-learning methods, decision trees and symbolic regression, and tools based on them, namely C5.0 and Eureqa Pro. Using the same training data we derive models to predict if to run the REPLICA architecture in PRAM or NUMA mode for a certain parameterized computation type (parallel stencil operation). Without machine-learning it had not been possible to derive predictors of when to use PRAM or NUMA mode.

The best predictors give a misclassification rate of 5%. Combining the three best ones using a majority voting algorithm misclassifications can be eliminated fully, at least on the test case. Average gained speedup over PRAM mode execution only ranges between 1.92 and 2.23 for all classifiers on the test cases.

For C5.0 it seems that adding more training data improves the accuracy while for Eureqa Pro more training data can generate more instable models. However, Eureqa Pro can be as good as C5.0 if the right training data is used and then it makes correct classification for the case where all other predictors are wrong. The derived formulas for PRAM and NUMA execution time are not accurate enough to predict the execution time, however they are accurate enough to be used for deciding if PRAM or NUMA mode should be used.

All the derived predictors are very simple, and can be implemented with a few computation and comparison instructions. The overhead of invoking them at runtime, if some parameters such as size are unknown statically, will be very small. As far as we know, Eureqa Pro has not been used for this type of performance predictions before.

Future work includes using the same methods on other problem types than stencil-like algorithms. It would als be interesting to test this on heterogeneous systems such as CPU-GPU based ones. Another interesting problem would be to derive parameterized models of algorithms using a pattern matching framework such as PRT [19] and combine it with machine learning. Each pattern could then be annotated with a predictor for the pattern implementations' best performance for specific parameters and for a given type of hardware.

Acknowledgments. This research is supported by SeRC. Thanks to Nutonian providing a free academic licence of Eureqa Pro. Thanks to Martti Forsell and Lu Li for their help and comments.

References

1. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A language and compiler for algorithmic choice. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland (June 2009)
2. Danylenko, A., Kessler, C., Löwe, W.: Comparing machine learning approaches for context-aware composition. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 18–33. Springer, Heidelberg (2011)

3. Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In: Proceedings of the 4th International Workshop on Multicore Software Engineering, IWMSE 2011, pp. 25–32. ACM, New York (2011)
4. Duesterwald, E., Gupta, R., Soffa, M.: Register pipelining: An integrated approach to register allocation for scalar and subscripted variables. In: Kastens, U., Pfahler, P. (eds.) CC 1992. LNCS, vol. 641, pp. 192–206. Springer, Heidelberg (1992)
5. Forsell, M.: Realizing multioperations for step cached MP-SOCs. In: Proc. SOC 2006 (2006)
6. Forsell, M.: Configurable Emulated Shared Memory Architecture for General Purpose MP-SoCs and NoC Regions. In: Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, San Diego, USA, May 10-13, pp. 163–172 (2009)
7. Forsell, M.: A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. In: Proceedings of the 12th Workshop on Advances in Parallel and Distributed Computational Models (in conjunction with the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2010), Atlanta, USA, April 19, pp. 1–8 (2010)
8. Forsell, M., Hansson, E., Kessler, C., Mäkelä, J.M., Leppänen, V.: NUMA Computing with Hardware and Software Co-support on Configurable Emulated Shared memory Architectures. International Journal of Networking and Computing 4(1) (2014)
9. Goel, B.: Per-core Power Estimation and Power Aware Scheduling Strategies for CMPs, 70 (2011)
10. Grewe, D., O'Boyle, M.: A static task partitioning approach for heterogeneous systems using opencl. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 286–305. Springer, Heidelberg (2011)
11. Hansson, E., Alnervik, E., Kessler, C., Forsell, M.: A Quantitative Comparison of PRAM based Emulated Shared Memory Architectures to Current Multicore CPUs and GPUs. In: 2014 27th International Conference on Architecture of Computing Systems (ARCS), pp. 1–7 (February 2014)
12. Keller, J., Kessler, C., Träff, J.L.: Practical PRAM Programming. John Wiley & Sons, Inc., New York (2001)
13. Kessler, M., Hansson, E., Åkesson, D., Kessler, C.: Exploiting instruction level parallelism for REPLICA - a configurable VLIW architecture with chained functional units. In: Proc. PDPTA 2012 (July 2012)
14. Li, L., Dastgeer, U., Kessler, C.: Pruning strategies in adaptive off-line tuning for optimized composition of components on heterogeneous systems. Accepted for Proc. Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP (2014)
15. Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., Amarasinghe, S.: Portable performance on heterogeneous architectures. SIGPLAN Not. 48(4), 431–444 (2013)
16. Quinlan, R.: C5.0 release 2.07 GPL Edition [software], `http://www.rulequest.com/download.html`
17. Schmidt, M., Lipson, H.: Eureqa (version 0.99.5 beta) [software] (2014), `http://www.nutonian.com`

18. Schmidt, M., Lipson, H.: Distilling Free-Form Natural Laws from Experimental Data. Science 324(5923), 81–85 (2009)
19. Shafiee Sarvestani, A., Hansson, E., Kessler, C.: Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. International Journal of Parallel Programming, 1–19 (2012)
20. Wernsing, J.R., Stitt, G.: Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. SIGPLAN Not. 45(4), 115–124 (2010)