2009 15th International Conference on Parallel and Distributed Systems

Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning

Xuan Chen
Department of Computer Science, JiNan University
Guangzhou 510632, P.R.China
e-mail: tcx@jnu.edu.cn

Shun Long
Department of Computer Science, JiNan University
Guangzhou 510632, P.R.China
e-mail: tlongshun@jnu.edu.cn

***Abstract*— The introduction of multi-core architectures generates a higher demand for parallelism in order to fully exploit the potential of modern computers. It is of vital importance that a compiler can allocate parallel workload in a cost-aware manner in order to achieve optimal performance on a multi-core architecture. This paper presents an adaptive OpenMP-based mechanism capable of generating a reasonable number of representative multi-threaded versions for a given loop, and selecting at runtime a suitable version to execute on a multi-core architecture. Preliminary experimental results show that, on average, it achieves 87% of the highest performance improvement across a whole spectrum of input sizes on two multi-core platforms.**

Keywords: parallelization; multi-versioning; machine learning; OpenMP

I. INTRODUCTION

The past decade has seen major chip manufacturers turning their focus from making one processor run faster to the development of multi-core architectures, in which multiple processors are placed on the same chip, communicating via hardware channels and shared memory. This architecture has generated a new demand for techniques that can fully exploit the architectural potential.

Parallelism[9] is one of the main sources of performance improvement in modern computing environments. However, it does not guarantee the most efficient use of shared memory, nor even performance improvement. Prior experience with multi-threaded Java[11] shows that when the workload is improperly shared among too many number of threads, the extra cost to create and synchronize them will offset the performance improvement achieved via parallelization. In many new application domains, this cost becomes non-negligible when compared to workload, or even results in performance degradation instead of improvement. Therefore, it is of vital importance that, when given a program, a compiler can adaptively allocate workload among multiple threads in order to achieve optimal performance in a multi-core environment.

OpenMP[19] is an industrial standard API that supports explicitly multi-threaded, shared memory parallelism among a variety of shared memory architectures and platforms. It offers programmers full control over parallelization via compiler directives, runtime libraries as well as environment.

This paper presents an adaptive mechanism which can generate for a given loop a reasonable number of representative OpenMP versions, and select at runtime which one to execute based on the runtime context. Preliminary experimental results show that it can efficiently allocate the workload among a suitable set of parallel threads and achieve optimal performance.

The outline of this paper is as follows. Section II presents the motivation of our work. The mechanism is explained in section III, before details are explained in section IV and V respectively. Preliminary experimental results are then presented in section VI, followed by a review of related works in section VII. A discussion about future work is given in section VIII, before some concluding remarks in section IX.

II. MOTIVATION

OpenMP[19] uses a fork-join model of parallel execution. It provides an directive omp_set_num_threads(numthreads) for programmers to explicitly specify/alter the number of team threads to be used in parallel regions. If not explicitly specified, the compiler will detect the hardware configuration, keep one core for the master thread, and generate one thread for each of the remaining cores within the processor.

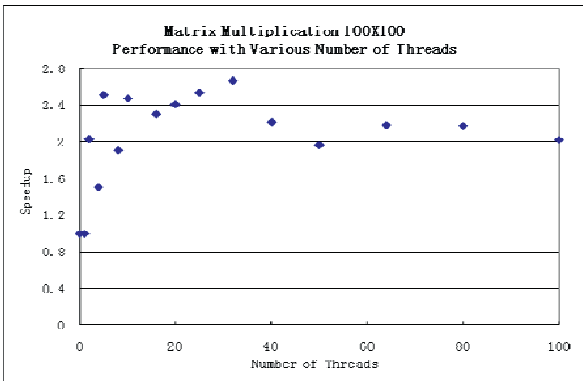

Figure.1 The performance of 100x100 matrix multiplication under various numbers of threads.

However, OpenMP does not necessarily guarantee the most efficient use of shared memory, i.e. the introduction of multiple team threads does not necessarily result in better performance. For instance, we ran a 100x100 matrix multiplication on a platform containing a 1GHz AMD Athlon(tm) 64 X2 Dual Core Processor 3600+ and 1G RAM, with gcc x86_64-linux-gnu 4.3, running under Ubuntu Linux 4.3.2-1ubuntu12 (kernel 2.6.27-14-generic). Different

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.77

versions with various numbers of threads (1, 2, 4, 5, 8, 10, 16, 20, 25, 32, 40, 50, 64, 80 and 100 respectively) were tested and the resulting speedups were then plotted against the number of threads, as shown in Figure.1. It shows that, generally saying, the speedup is on a rise as the number of threads increases, i.e. higher speedups are obtained when there are more team threads sharing the workload. It reaches its peak (speedup = 2.67) when 32 threads are used. Nevertheless, no better performance can be achieved when even more team threads are used. Instead, the speedups drops, as the 40- and 50-thread cases demonstrate. This is because the extra cost to create and synchronize the additional team threads has offset the performance gain obtained via parallelization. More interestingly, although we presume a further performance drop when even more threads are used, this is actually not the case, since the performances of 64, 80 and 100 threads are not much lower than that of 50 threads.

The above example shows that different numbers of threads result in varied performances. We have observed similar performance variances on other programs and on different platforms, which shows that this phenomenon is program-, data- and platform-relevant. This suggests that a compiler shall not make this number-of-thread decision in a static manner. Instead, this decision should be made at runtime based on the runtime context.

III. AN ADAPTIVE MULTI-VERSIONING PARALLELIZATION MECHANISM

First, confirm that you have the correct template for your paper size. This template has been tailored for output on the US-letter paper size. If you are using A4-sized paper, please close this template and download the file for A4 paper format called "CPS_A4_format".

Given a specific program, it is difficult to precisely predict the best number of team threads for parallelization at compile time. Adaptive optimization [14] is therefore needed in order to make this decision based on runtime context.

Adaptive optimization is achieved via techniques such as dynamic compilation, and multi-versioning, etc. However, dynamic compilation needs extra time for runtime re-compilation of some code regions, which makes it not suitable in our cases. Multi-versioning is a reasonable approach because it is unlikely that any single static version can adapt and yield high performance across different runtime contexts. This motivates us to find an adaptive mechanism (as demonstrated in Figure.2) to generate representative versions and, at runtime, make version selection decision based on the runtime context.

Given a program, it shall generate only a reasonable number of representative versions in order to avoid code explosion, as explained later. In addition, a runtime decision making structure should be constructed (as the switch structure in Figure.2(B)) in order to decide at run-time which version to execute in order to achieve higher performance. It is worth noting that this framework as well as most of the codes can be implemented as a static code template.

Previous related works indicated that making this version selection decision at runtime is not straightforward. However, they found that programs with similar workload are likely to

```
// the sequential version
for (i=0; i<N; i++)
   … …  // loop body
```

(A) a given loop in its original sequential form

```
// the OpenMP parallel version
switch (certain conditions) {
  case …: // the tn0 thread version
    #ifndef NOOMP
    omp_set_num_threads(tn0);
    #pragma omp parallel default(none)
    {  #pragma omp for
       #endif
       for (i=0;i<N;i++)
          … … // loop body
    }  break;
  … …
  //  more OpenMP version with various numbers
  //  of threads
  case …: // the tnv-1 thread version
    #ifndef NOOMP
    omp_set_num_threads(tnv-1);
    #pragma omp parallel default(none)
    {  #pragma omp for
       #endif
       for (i=0;i<N;i++)
          … …  // loop body
    }  break;
  … …
  default:  // the tnv thread version
    #ifndef NOOMP
    omp_set_num_threads(tnv);
    #pragma omp parallel default(none)
    {  #pragma omp for
       #endif
       for (i=0;i<N;i++)
          … …  // loop body
    } break;
  }
```

(B) its adaptive multi-versioned OpenMP version

Figure.2 Illustration of our proposed framework, which turns a sequential loop (A) into a multi-versioning OpenMP-parallelized equivalent (B)

benefit from the same or similar parallelization scheme[11]. A compiler can exploit this observation and make the parallelization decision based on its previous experience with similar programs.

Machine learning[13] is a natural approach to exploit such similarities. We choose instance-based learning and use features of a program/loop to make an implicit estimate of its workload. They are not only easy to capture but also sufficient for training purpose, whilst an explicit estimate is more difficult to obtain.

Our proposed adaptive multi-versioning mechanism works in the following manner. When a program/loop is encountered, the compiler first generates a reasonable number of versions (each with a different number of threads)

in an iterative manner. These versions are then evaluated with inputs of various sizes. Then, based on the performance feedback, the compiler selects a small number of candidate parallel versions for the final executable, before the code for runtime version selection is also generated and embedded into the executable, which maps the features (of both program and runtime input) to versions.

IV. GENERATION OF CANDIDATE VERSIONS

By using the *omp_set_num_threads(...)* directive to specify various numbers of threads, a compiler can generate for a given loop as many versions as necessary. Previous experiments show that these different versions usually result in various performances. In order to maximize the mean performance across all possible inputs, the compiler shall consider only those of good performance as the candidates. Furthermore, because the inclusion of too many versions will lead to code explosion, it shall pick only a reasonable number of them from all possible options. In addition, these candidates are expected to be representative, in that they could achieve high performance across the whole spectrum of possible inputs. Decisions must be made in order to balance these three concerns discussed above.

A heuristic approach (as in Figure.3) is developed to select representative versions for a given program/loop. First, a set of parallel versions are generated for testing purpose, each with a different numbers of threads. They are then evaluated with various inputs of selected input sizes *sz*s. Each test case is specified by the static code features such as loop nest depth, number of arrays used and the others, as well as the dynamic feature in data set size. The corresponding performances (speedup) ($P_{f,tn}$s) are recorded together with both the program feature vectors *f*s and the corresponding thread numbers t_ns, in the form of a triple (f, t_n, $P_{f,tn}$). Let BP_f be the highest speedup achieved for each *f*. It is considered as the best among all possible parallel versions. The efficiency of each version $E_{f,tn} = P_{f,tn}/BP_f$ is therefore be obtained.

Next, we calculate each t_n's profitability across different program versions in search of t_ns that can bring high performance across various input sizes. It is worth noting that, as we cannot completely eliminate the impact of noise or other unknown factors on performance (as shown in Figure.1), the compiler considers the t_ns that yield similar performance equivalent and tends to pick the smallest tn from them. A simple credit system is developed to award each t_n certain credits if its efficiency reaches a certain level on a certain test case. For instance, if the efficiency of a t_n is 95% or higher in one test case, it is awarded five points, and it is awarded another four points if its efficiency for another case is between 90% and 95%, etc. A sorted list L of t_ns is then obtained by sorting them in descending order of their credits/profitability.

Finally, we select from top of list *L* a certain number of thread numbers. The selection algorithm in Figure.3 shows that, for a given loop, the compiler can create *v* candidate versions, each corresponds to a parallel version with each of these selected thread number t_ns.

```
// P: the program to be parallelized
// v: a predefined no. of candidate versions
// s: the set of thread numbers selected
generate versions of P by parallelizing it with
    various numbers of threads tns;
test-run each these versions Pis with inputs of
    various sizes, and record their performances;
for each input size {
  find the best performance BP;
}
for each test-run {
  calculate each version's efficiency;
  if (its efficiency reaches a certain level)
    award the corresponding tn a certain points;
}
sort all tns in a temporary list L according to their
    points awarded
s = {}; count =0;
repeat {
  pick the first thread number tn from L;
  if (tn is not too close to any element in s) {
     s = s+{tn}; count++;
  }
} until (count ==v);
generate parallel versions of P according to
  the thread numbers selected in s;
```

Figure.3 Pseudo code of the candidate selection algorithm

It is worth noting that we do not necessarily pick from *L* the t_n that gives the very best performance, if its neighboring t_ns have already been selected, as demonstrated by the *if*-statement with the condition (t_n not too close to any element in *s*) in Figure.3. The motivation/rationality behind this heuristic is that we hope to prevent the candidate versions aggregate within a small spectrum, this helps improve the representativeness of these candidate versions and, in turn, coverage and applicability of the executable.

Both thresholds $E_{threshold}$ and *v* are currently specified as compile-time parameters in the –Ox form in our prototype system. Fine-tuning of these parameters and the credit system is left to future work.

V. VERSION SELECTION FRAMEWORK

Prior research in learning based optimizing compiler [1][4][5][11][12] use various static or dynamic features to reveal important details of a given program and to make an implicit estimate of its workload. Considering the loop parallelization problem described above, we consider only static loop-related features and associate them with various parallel scheme (numbers of threads used) and the corresponding performance (speedups achieved), as explained in the previous section.

In-depth analysis[11][12] shows that loop size is the dominating factor among them and is sufficient for our version selection purpose. It also suggests that the number of features might not be as important as the distribution of the

values of them[12]. Therefore, we use size of the outmost loop as the only feature to estimate similarities among loops.

The generation of the version selection code is relatively straight forward. K-nearest neighbor algorithm is used in our adaptive mechanism, which associates each of the thread numbers t_{ni}s selected in the previous section with a specific loop size ls_i. When a loop is encountered at runtime, its size is captured and compared with all the t_{ni}s. If it is identical or closest to a particular tn_k, then the corresponding version v_k is selected for execution. The resulting switch-like code structure is generated via a predefined code template, as shown in Figure.2(B), before being embedded in the executable.

VI. Preliminary Experimental Results

A. Experiment Setup

We evaluated the proposed mechanism in two different environments. One is the AMD-Athlon environment as specified in section 2, the other is a Dell PowerEdge 2950 server which contains an Intel QuadCore(tm) processor with four 2GHz Xeon E5405 cores and 2G RAM, and the compiler is gcc x86_64-suse-linux version 4.3.2, running under openSUSE Linux 2.6.27.7-9-default.

Two programs (one numeric and the other non-numeric) are used in our preliminary experiments. One is the matrix multiplication widely used in traditional high performance computing, with various data sizes (matrix sizes) from 100 to 2000, denoted respectively as MM100, .. MM2000 etc. The other is TF-IDF used in information retrieval[10], which calculates the term frequencies and inverse document frequencies before calculating the vectors of a list of documents. It uses a vocabulary of size 2000 and data sizes (numbers of documents) vary from 100 to 2000, denoted as TFIDF100, … TFIDF2000 respectively.

The experiments are carried out in the following manner. Take MM as an example. First, we select three data sizes (one small (MM200), one medium (MM800) and one large(MM1500)) and test-run them with various numbers of threads. The results are used to train the compiler as discussed before. Once trained, the compiler generates a multi-versioned MM executable. To keep the code size modest, the compiler generates only a three-versioned one, and evaluates it with data sets of various sizes. Its performance is then compared against that of a random algorithm. The results of experiments carried out on the above two platforms are summarized in Table I to IV respectively.

B. Results on Matrix Multiplication

Table I and II show that, on both platforms, the adaptive mechanism outperforms its random counterpart in all but one test cases with data sizes vary from 100 to2000. For instance, on the AMD-Athlon platform, for all the eleven cases, the efficiencies are all above 90%. Particularly, they are even higher at 95% or above in seven of them, indicating that the adaptive mechanism is capable of identifying the sub-optimal parallel schemes for data sets of various sizes. We believe this is due to the fact that, once properly trained, the compiler can make a more precise estimation of the workload based on runtime profile collected from test-runs, instead of based on a predictive model.

TABLE I. Performance comparison between the proposed adaptive mechanism and a random mechanism for Matrix Multiplication on AMD-Athlon. The proposed adaptive mechanism outperforms the latter by 11%.

Program	Random		Learning	
	speedup	*%*	*speedup*	*%*
MM100	2.13	80%	2.47	93%
MM300	2.36	85%	2.61	93%
MM400	1.74	84%	1.90	91%
MM500	1.63	85%	1.78	92%
MM600	1.69	86%	1.92	97%
MM700	1.75	86%	2.04	99%
MM900	1.68	86%	1.92	99%
MM1000	1.59	78%	1.83	99%
MM1200	1.59	86%	1.82	99%
MM1800	1.56	87%	1.75	98%
MM2000	1.54	87%	1.73	98%
Average		85%		96%

TABLE II. Performance comparison between the proposed adaptive mechanism and a random mechanism for Matrix Multiplication on Intel-QuadCore. The proposed adaptive mechanism outperforms the latter by 8%.

Program	Random		Learning	
	speedup	*%*	*speedup*	*%*
MM100	1.66	76%	1.46	67%
MM300	2.74	84%	3.12	96%
MM400	3.07	82%	3.63	98%
MM500	2.95	87%	3.23	95%
MM600	3.07	88%	3.41	99%
MM700	3.10	89%	3.44	99%
MM900	3.15	91%	3.48	100%
MM1000	3.12	89%	3.49	100%
MM1200	3.31	92%	3.53	99%
MM1800	3.43	89%	3.74	97%
MM2000	3.44	91%	3.73	99%
Average		87%		95%

Similar performance can also be found on the Intel-QuadCore platform, where our learning based mechanism reaches an average efficiency of 95%, compared to that of 87% from the random selection mechanism. It is worth noting that the random mechanism outperforms our mechanism in MM100 which has a small data set. In-depth look at the raw profile (Figure.1) suggests that the performance of MM is very sensitive to the number of team threads used when the data set is of small, as also suggested in many related research. On average, our adaptive multi-versioning mechanism achieves 96% and 95% of the highest performance improvement across all eleven programs on these two platforms. Similar performances have also been achieved if we pick training cases in a similar manner.

On average, our adaptive multi-versioning mechanism achieves 96% and 95% of the highest performance improvement across all eleven programs on these two platforms. Similar performances have also been achieved if we pick training cases in a similar manner.

TABLE III. PERFORMANCE COMPARISON BETWEEN THE PROPOSED ADAPTIVE MECHANISM AND A RANDOM MECHANISM FOR TF-IDF ON AMD-ATHLON. THE PROPOSED ADAPTIVE MECHANISM OUTPERFORMS THE LATTER BY 10%.

Program	Random		Learning	
	speedup	*%*	*speedup*	*%*
TFIDF100	-	-	-	-
TFIDF300	-	-	-	-
TFIDF400	1.09	68%	1.29	80%
TFIDF500	1.15	71%	1.35	84%
TFIDF600	1.18	79%	1.42	95%
TFIDF700	1.29	73%	1.34	76%
TFIDF900	1.41	81%	1.73	100%
TFIDF1000	1.38	79%	1.58	90%
TFIDF1200	2.13	84%	2.32	92%
TFIDF1800	2.13	82%	2.60	100%
TFIDF2000	1.57	77%	1.63	80%
Average		62%		72%

TABLE IV. PERFORMANCE COMPARISON BETWEEN THE PROPOSED ADAPTIVE MECHANISM AND A RANDOM MECHANISM FOR TF-IDF ON INTEL-QUADCORE. THE PROPOSED ADAPTIVE MECHANISM OUTPERFORMS THE LATTER BY 9%.

Program	Random		Learning	
	speedup	*%*	*speedup*	*%*
TFIDF100	-	-	-	-
TFIDF300	1.51	75%	1.66	83%
TFIDF400	1.57	78%	1.68	83%
TFIDF500	1.50	75%	1.90	95%
TFIDF600	1.59	86%	1.75	95%
TFIDF700	1.53	81%	1.81	97%
TFIDF900	1.60	86%	1.83	98%
TFIDF1000	1.60	86%	1.74	93%
TFIDF1200	1.64	88%	1.74	93%
TFIDF1800	1.64	89%	1.79	96%
TFIDF2000	1.65	89%	1.79	96%
Average		75%		84%

C. *Results on TFIDF*

Table III and IV show that similar results have been found for TFIDF on both platforms. Our adaptive mechanism outperforms the random algorithm by 10% and 9% respectively. However, the OpenMP schemes chosen for TFIDF100 and 300 (on AMD-Athlon) and TFIDF100 (on Intel-QuadCore) provide no performance improvement. The raw profile shows that no scheme except the 4-thread one can improve the performance of TF-IDF100. But our mechanism chose a 2-thread scheme instead, based on the profile collected from TFIDF200. This could also explain the failure of our mechanism for TFIDF300 on AMD-Athlon, and that for TFIDF100 on both platforms. On average, our adaptive multi-versioning mechanism achieves only 72% and 84% of the highest performance improvement across all 11 programs on these two platforms.

On average, our adaptive multi-versioning mechanism achieves only 72% and 84% of the highest performance improvement across all 11 programs on these two platforms. Both are lower than those of the matrix multiplication cases, mainly because of the three no-improvement cases explained above. This suggests that further improvement should be made to deal with data of smaller sizes.

Furthermore, we have also applied the above learning results to a new TFIDF with an even larger vocabulary of 3000 and 5000 respectively. The results show that, on average, it achieved 93% of the highest speedups across input sizes between 6000 and 10000. This demonstrates the applicability of our mechanism across an even larger data spectrum. It is also worth noting that, for MM and TFIDF, the results learned from the AMD-Athlon platform are very similar to that from the Intel-QuadCore one, which hints portability to a certain extent.

VII. RELATED WORK

There is a rich literature about parallelism[9], covering topics from parallel compiler[1] to architectural supports[6]. Blume et.al.[3] gives a comprehensive review of the state of the art in this area, as well as a good description of the challenges.

Wang.et.al.[16] presents two predictors based on artificial neural network and support vector machine. They can use a model learned offline to select the best mapping (including the number of threads and the scheduling policy) for parallel programs on multi-core processors. This is very similar to our mechanism. The main differences are: 1) they choose the dynamic compilation approach whilst our work uses multi-versioning; 2) they use machine learning not only to model a machine's behavior but also to predict the best number of threads and the scheduling policy for a given program; 3) they use 6 features (3 about code and 3 about data and runtime performance) whilst we use only one in size of the outmost loop (as explained above). 4) they use artificial neural network to solve the scalability problem and a support vector machine model to solve the scheduling policy classification problem, both approaches known to be time-consuming in training. On the contrary, we use a much simpler instance-based learning approach and achieve similar performance. 5) our mechanism generates a multi-versioning executable valid across all input sizes, whilst they to use the pre-built model to decide at runtime the best number of threads; and finally, we have not tackled the problem of mapping a given parallel program to a platform as they have done.

Tournavitis et.al.[15] further improves Wang's work in developing a profile-driven approach which is capable of not only identifying potential parallelisms but also mapping them on a given platform. Machine learning techniques are used to make better mapping decision and provide more scope for adaptation to different target architectures.

There are some other related works which develop heuristics, analytical and feedback direct models in order to achieve adaptive task scheduling. Corbalan etl.al.[8] proposed an adaptive loop scheduler which selects both thread numbers and scheduling policy for a parallel region in SMPs based on feedback-directed runtime decisions[14]. Blagojevic et.al.[2] presents an approach to allocate processor for loops at runtime. An analytical model is proposed in [18] which use program and architectural

information to model a parallel program. These models are inevitably architecture-specific and therefore not portable. Xekelakis et.al.[17] combines three multi-threaded execution models (thread level speculation, helper threads and run-ahead execution) into a single one and single hardware infrastructure. It results in an adaptive system which can find the most appropriate execution model for a given program at runtime.

Machine learning[13] has recently been introduced to compiler optimization at system level. Various approaches are used in iterative optimization[1][7] to explore a large optimization space. [4] builds a performance model based on only a small number of evaluations, which significantly reduces the cost of evaluating the impact of compiler optimizations. Logistic regression is used in [5] to derive a predictive model that selects suitable optimizations to apply to each method based on code features. [11] use instance-based learning to select the most promising workload allocation scheme for a Java program and it does not generate multiple versions for runtime selection. Wang.et.al.[16] presents two predictors based on artificial neural network and support vector machine. They can use a model learned offline to select the best mapping (including the number of threads and the scheduling policy) for parallel programs on multi-core processors.

VIII. CONCLUSIONS

This paper presents an adaptive mechanism which, when given a loop, can generate a reasonable number of representative OpenMP versions, and select at runtime which one to execute based on the runtime context. Preliminary experimental results show that, on average, it achieves 87% of the highest performance improvement on two different platforms, compared to 77% of a random selection algorithm.

Further improvement could be made for the purposed mechanism. For instance, the adoption of low-cost profiling techniques could lower the cost of iterative evaluation of various code versions. Machine learning techniques could be used to select representative code sizes for testing. Furthermore, additional code features could provide useful hint to the workload of each iteration so that the learning results could benefit not only the current program-to-compile, but also all the programs encountered in the future. Machine learning techniques such as PCA could be used to identify good features from all the potential candidates, in order to keep a proper balance between the efficiency of learning and the number of features used. We are also working on the selection of proper threshold values (such as the number of versions to be generated), in order to further lower the cost of compilation without loss in efficiency.

REFERENCES

[1] F.Agakov, E.Bonilla, J.Cavazos, B.Franke, G.Fursin, M.F.P.O'Boyle，J.Thomson, M.Toussaint and C.Williams, "Using machine learning to focus iterative optimization," Proc. of the 2006 International Symposium on Code Generation and Optimization (CGO'06), 2006

[2] F.Blagojevic, X.Feng, K.Cameron and D.S.Nikolopoulos, "Modeling multi-grain parallelism on heterogeneous multicore processors: a case study of the Cell BE," Proc. of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC'08). 2008.

[3] W.Blume, R.Eigenmann, J.Hoeflinger, D.Padua, L.Rauchwerger and T.Peng, "Automatic detection of parallelism, a grand challenge for high performance computing," IEEE Parallel and Distributed Technology, 2(3), 1994.

[4] J.Cavazos, C.Dubach, F.Agakov, E.Bonilla, M.O'Boyle, G.Fursin and O.Temam, "Automatic performance model construction for the fast software exploration of new hardware design," Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06), 2006.

[5] J.Cavazos and M.F.P.O'Boyle, "Method-specific dynamic compilation using logistic regression," Proc. of ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06), 2006.

[6] M.Cintra, J.Martinez, and J.Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," Proc. of the Intl. Symp. on Computer Architecture (ISCA), 2000.

[7] K.Cooper, D.Subranmanian and L.Torzon, "Adaptive optimizing compilers for the 21st century," Journal of Supercomputing, 23(1), 2001.

[8] J.Corbalan, X.Martorell and J.Labarta, "Performance driven processor allocation," IEEE Transactions Parallel Distribution System, 16(7). 2005.

[9] J.Dongarra, I.Foster, G.Fox, K.Kennedy, W.Gropp, L.Torczon and A.White, Sourcebook of parallel computing, Morgan Kaufmann, US, 2003.

[10] D.A.Grossman and O.Frieder, Information Retrieval, Algorithms and Heuristics (2nd ed), Springer, 2004.

[11] S.Long, G.Fursin and B.Franke, "A cost-aware parallel workload allocation approach based on machine learning techniques," Proc. of the IFIP International Conference on Network and Parallel Computing (NPC) , 2007.

[12] L.Luo, Y.Chen, C.Wu, S.Long and G.Fursin, "Finding representative sets of optimizations for adaptive multiversioning applications," Proc. of the 3rd Workshop on Statistical and Machine learning approaches to Architecture and compilaTion (SMART'09), 2009

[13] T.Mitchell, Machine learning, McGraw-Hill, US, 1997.

[14] M.Smith, "Overcoming the challenges to feedback-directed optimization," Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), 2000.

[15] G.Tournavitis, Z.Wang, B.Franke and M.O'Boyle. "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," Proc. of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09), 2009.

[16] Z.Wang and M.F.P.O'Boyle, "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach," Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2009.

[17] P.Xekelakis, N.Ioannou and M.Cintra, "Combining thread level speculation, helper threads and runahead execution," Proc. of the 2009 International Conference on Supercomputing (ICS09), 2009.

[18] Z.Yun and V.Michael, "Runtime empirical selection of loop shcedulers on hyperthreaded SMPs," Proc. of 2005 IEEE International Parallel & Distributed Processing Symposium (IPDPS'05). 2005

[19] OpenMP homepage, www.openmp.org.