A Cost-Aware Parallel Workload Allocation Approach based on Machine Learning Techniques

Shun Long[1], Grigori Fursin[2], Björn Franke[3]

[1] Department of Computer Science, Jinan University, Guangzhou 510632, P.R.China *{long.shun@gmail.com}*
[2] Member of HiPEAC, ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France *{grigori.fursin@inria.fr}*
[3] Member of HiPEAC, Institute for Computing Systems Architecture, The University of Edinburgh, UK *{bfranke@inf.ed.ac.uk}*

Abstract. Parallelism is one of the main sources for performance improvement in modern computing environment, but the efficient exploitation of the available parallelism depends on a number of parameters. Determining the optimum number of threads for a given data parallel loop, for example, is a difficult problem and dependent on the specific parallel platform. This paper presents a learning-based approach to parallel workload allocation in a cost-aware manner. This approach uses static program features to classify programs, before deciding the best workload allocation scheme based on its prior experience with similar programs. Experimental results on 12 Java benchmarks (76 test cases with different workloads in total) show that it can efficiently allocate the parallel workload among Java threads and achieve an efficiency of 86% on average.

Keywords: parallelism, workload allocation, cost, instance-based learning.

1 Introduction

Parallelism is one of the main sources for performance improvement [4][11] in modern computing environment. This is particularly true in the area of high performance computing, where the cost of parallelism (on thread creation, scheduling and communication) is usually negligible when compared to heavy workload. However, the rapidly evolving hardware technology enables parallelization in most modern computing systems, for instance embedded devices. In many of these systems, the cost of parallelization becomes non-negligible when compared to workload. Moreover, inefficient workload allocation could even degrade the performance considerably, which is not acceptable. Therefore, it is vitally important to allocate the workload in a cost-aware manner in order to achieve optimal performance.

Java is a widely used programming language with multi-threading features. But the development of efficient parallel Java programs requires careful consideration to avoid performance degradation due to the cost of thread creation, scheduling and communication. This cost depends on many environment-specific factors (CPU, cache, memory, operating system, Java virtual machine, etc). The interaction of these factors is hard to model or predict in advance. In search for optimal performance in a given execution environment, it is expected that the compiler or virtual machines uses an adaptive workload allocation approach, so that the workload can be allocated among Java threads in a cost-aware manner. This can be achieved statically or in a dynamic manner via speculative parallelization, both approaches have their specific advantages and disadvantages as discussed in [12].

This paper presents a cost-aware parallel workload allocation approach based on machine learning techniques [15]. It learns from training examples how to allocate parallel workload among Java threads. When a new program is encountered, the compiler extracts its static program features for classification purpose, retrieves its prior experience with similar training examples, and uses this knowledge to decide the best parallel scheme for the new program. Experimental results suggest that this approach can effectively allocate workload among various numbers of threads and achieve optimal or sub-optimal performance.

The outline of this paper is as follows. Section 2 presents a Java multi-threaded framework, before demonstrating via some preliminary experiments the demand for a cost-aware parallel workload allocation approach. Section 3 presents our adaptive approach. Section 4 evaluates its performance and analyzes the results. Section 5 briefly reviews some related work, before some concluding remarks in section 6.

2 Motivation

We first present a Java multi-threaded framework which uses the class ExecutorService in package java.util.concurrent to parallelize a given *for* loop. For example, the sequential loop presented on the left of Table.1 can be parallelized as that on the right. The modification to the original code is given in bold font. Most of these modifications are loop-irrelevant and therefore can be implemented as a code template in advance. When a for loop is encountered, the compiler replaces the loop with the code template, copies the loop body into the run method of an inner class *Task*, before embedding Task into the class. The only question remains to be solved is the parallel scheme, i.e. how many threads should be used to share the workload in order to achieve optimal performance. Because this experiment aims to evaluate how a loop's performance varies under different schemes. They are decided in advance.
This framework is evaluated on a platform containing dual Xeon processors (each with two 2.80GHz cores) and 4G RAM, with Java Platform Standard Edition 1.5.0_02 running under Redhat Linux 2.6.12-1.1372_FC3cmp. Twelve benchmarks are chosen from the Java version of the DSPStone benchmark suite [22]. In total, there are 76 test cases derived from these benchmarks, each containing one *for* loop of a specific workload. *benchmark_n* is used to label the test case of benchmark with a workload of 2^n , and parallel level *m* is used to denote the scheme which allocates the workload evenly among 2^m threads. For simplicity concern, the experiment only considers workloads and thread numbers proportional to 2, so that the workload can be evenly distributed among threads. The discussion about more general workload

Table 1. A sequential Java loop (*Example_Squential,*on the left) and its parallel version (*Example_Parallel*, on the right) via the Java multi-threading framework.

public class Example_Sequential {
int size;
public static void main(
  String[] args)
    throws InterruptedException {
    …
    for (int i = 0; i < size ; i++) {
      system.out.println(i)); }
       …
  }
  …
  }
	import java.util.concurrent.*;
	public class Example_Parallel {
	  static int size;
	  static int numberofThreads;
	  static ExecutorService service;
	  public static void main(String[] args)
	    throws InterruptedException {
	    …
	  Service =
	  Executors.newFixedThreadPool(
	    numberofThreads);
	  for (int i = 0; i<numberofThreads; i++)
	  { service.execute(new Task(i)); }
	  service.shutdown();
	        …
	  }
	  private static class Task
	    implements Runnable {
	    private final int id;
	    private int lbound, ubound;
	    public Task(int id) {
	      this.id = id;
	      lbound = id * size
	          / numberofThreads;
	      ubound=
	      (id==numberofThread-1) ? size :
	        (id+1)*size/numberofThreads;
	    }
	    public void run() {
	      for (int i=lbound; i<ubound; i++) {
	        system.out.println(i); **}**
	    }
	    …
	  }
	}

sharing and load imbalance is left to future work.

The impact of different parallel schemes on each benchmark is summarized in Fig.1. It demonstrates that parallelism can significantly improve the performance of most test cases. Take *Matrix_3* as an example, when parallel level *m* increases, there are more threads to share its workload. This results in a rising speedup, which reaches its highest (37.36) when the workload is shared among 2^5=32 threads. However, further increasing *m* means that more threads are created. This implies more time spent on thread creation and scheduling, which diminishes the performance improvement achieved via parallelism.

When applicable, all *for* loops in Java programs can be parallelized in the above manner. Due to different cost in thread creation, scheduling and communication, a program's performance may vary significantly when parallelized with different schemes (i.e. *numberofThreads*). In some cases, improper schemes may even degrade

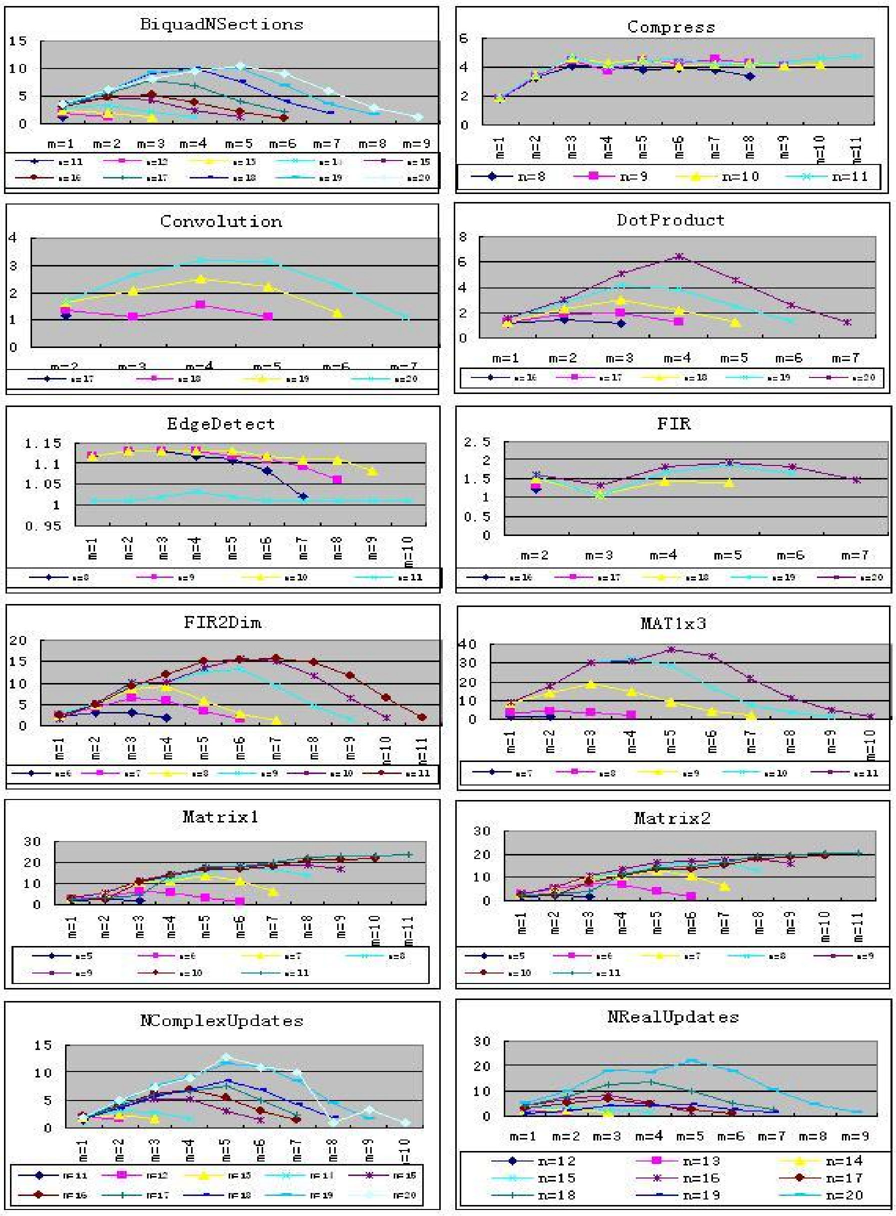

Fig.1. Performance improvement via parallelism, over 12 benchmarks with different workloads (2^n) and different workload allocation schemes (2^m), where the speedup achieved is plotted against the number of threads used to share the workload.

the performance. In search for higher performance via multi-threading, the compiler or virtual machine should decide the scheme in a cost-aware manner.

Analysis of experimental results reveals that *programs with similar workloads are likely to benefit from the same or similar parallel scheme*. This is best illustrated by *Matrix1* and *Matrix2* in Fig.1. Although coded in different manner, they actually do the same job, which results in performance curves almost identical. If good performance improvement can be achieved by using N threads to parallelize *Matrix1* of workload W, it is likely that a similar improvement can also be achieved by using N threads on *Matrix2* of the same workload, and vice versa.

This observation hints that a compiler could make the parallel scheme decision based on its previous experience with other programs. Each time a program is parallelized, the scheme and the resulting performance are stored in a database along with a description of the program. When a new program is encountered, this database is searched for programs similar to it. The best scheme for the most similar programs is then considered for the new one. This idea has long been used in static compilation analysis, which usually examines a few features of a program to see if it fits a model for a specific optimization.

3 Parallel Workload Allocation – Instance-Based Learning

Machine learning [15] is a natural approach to exploit such similarities. There are many machine learning approaches available. They vary in cost, complexity, efficiency and applicability. We believe that instance-based learning fits our objectives in effectiveness, timeliness and applicability better. Therefore, a Parallel Workload Allocation approach based on Instance-Based Learning (PWA-IBL) is developed.

PWA-IBL is based on the above observation that *programs with similar workload are likely to benefit from the same or similar parallel scheme*. It allocates the workload of a loop based on its previous experience with other similar loops. PWA-IBL consists of two steps: first it learns parallel schemes by being trained with a set of examples either carefully or randomly chosen; then it applies the knowledge when a new loop is encountered.

During the training phase, each time a loop is parallelized, its features are captured for classification. This can be considered as making an implicit estimate of its workload. Such an implicit estimate is not only easier to make but also sufficient for our purpose (as demonstrated later), whilst an explicit estimate is more difficult to achieve. A new category is created in the database if none of the existing ones has a similar workload. Then, the loop description is stored within the category, together with different parallel schemes and the corresponding performance improvement (for example, the resulting speedup or of other metrics).

When a new loop is encountered, PWA-IBL captures its features and classifies it. The most similar loop category within the database is identified. Then, based on its prior parallelization experience with the loops of this category, PWA-IBL selects the best scheme, before creating the threads and allocating the workload among them.

To implement PWA-IBL, the compiler must correlate loops, number of parallel threads and the resulting performance improvement in a systematic manner. In machine learning terms, the inputs or features of the problem are a description of the program and the workload allocation scheme, and the output is the performance improvement. These features not only reveal important details of the program but also help a compiler in classification. Therefore, they must be formally specified in order to enable instance-based learning.

PWA-IBL uses five program features for loop description and classification purposes. They are 1) loop depth; 2) loop size; 3) number of arrays used; 4) number of statements within the loop body, and 5) number of array references. It is understandable that not all program features play an equal role in workload estimation. Therefore, different weights are assigned to different features during classification, with higher weights given to feature 1), 2) and 4). Other features that can better describe the loop for PWA-IBL, but this set of features can be readily obtained from most compilers' internal representation of a program.

4 Evaluation

PWA-IBL is tested in the same environment as specified in section 2. The experiment aims to evaluate the impact of the training set selection on the performance of PWA-IBL, i.e. how enlarging the training set can improve its predictability. It is carried out in a cross-validation manner. For each *benchmark_n*, the size of training set is first decided, before the set itself is formed by selecting examples from the other 75 programs. For simplicity, only sizes 1, 5, 10, 15, 30, 45, 60 and 75 are considered, as it is unnecessary to increase the training set size by one each time. In addition, training examples are randomly chosen from all the other 75 programs. This is repeated for *t* times (*t*=30 in our test), each time with a different training set, before an average speedup is obtained.

For each *benchmark_n*, PWA-IBL applies its knowledge learned from a training set of a certain size *s*. If the chosen scheme cannot improve the performance, the resulting speedup is considered 0. Let *r* be the highest speedup achieved on *benchmark_n*. The performance of PWA-IBL with size *s* is defined as *r'/r*, where *r'* the average speedup achieved across *t* different training sets of size *s*. Take *DotProduct_20* as an example, when the training set size *s*=30, PWA-IBL achieves an average speedup of 5.36 over *t*=30 evaluations. Fig.1 shows that the highest speedup for *DotProduct_20* is 6.46 (when its workload is shared among 2^4=16 threads), i.e. the performance of PWA-IBL is 5.36/6.46=83% with the training set size *s*=30.

Fig.2 summarizes the performance of PWA-IBL on all 76 test cases, where it is plotted against the training set size *s*. Take *DotProduct_20* as an example, when the training set size *s*=1, PWA-IBL is equivalent to a random algorithm, as it learns only from one randomly selected example. The average speedup it achieves is 3.81, and the resulting performance is 3.81/6.46 =59%. When *s*=5, the average speedup is 4.07 and its performance is 63%, and so on. When *s*=75, PWA-IBL learns from all the other 75 test cases. This is equivalent to the "leave one out" cross validation test. The best scheme (share the workload among 2^4= 16 threads) from the one most similar to *DotProduct_20* is selected, resulting in a speedup of 6.46, when its performance reaches 100%. Similar results are found on most curves in Fig.2, which shows that as the training set size increases, the performance of PWA-IBL improves accordingly,

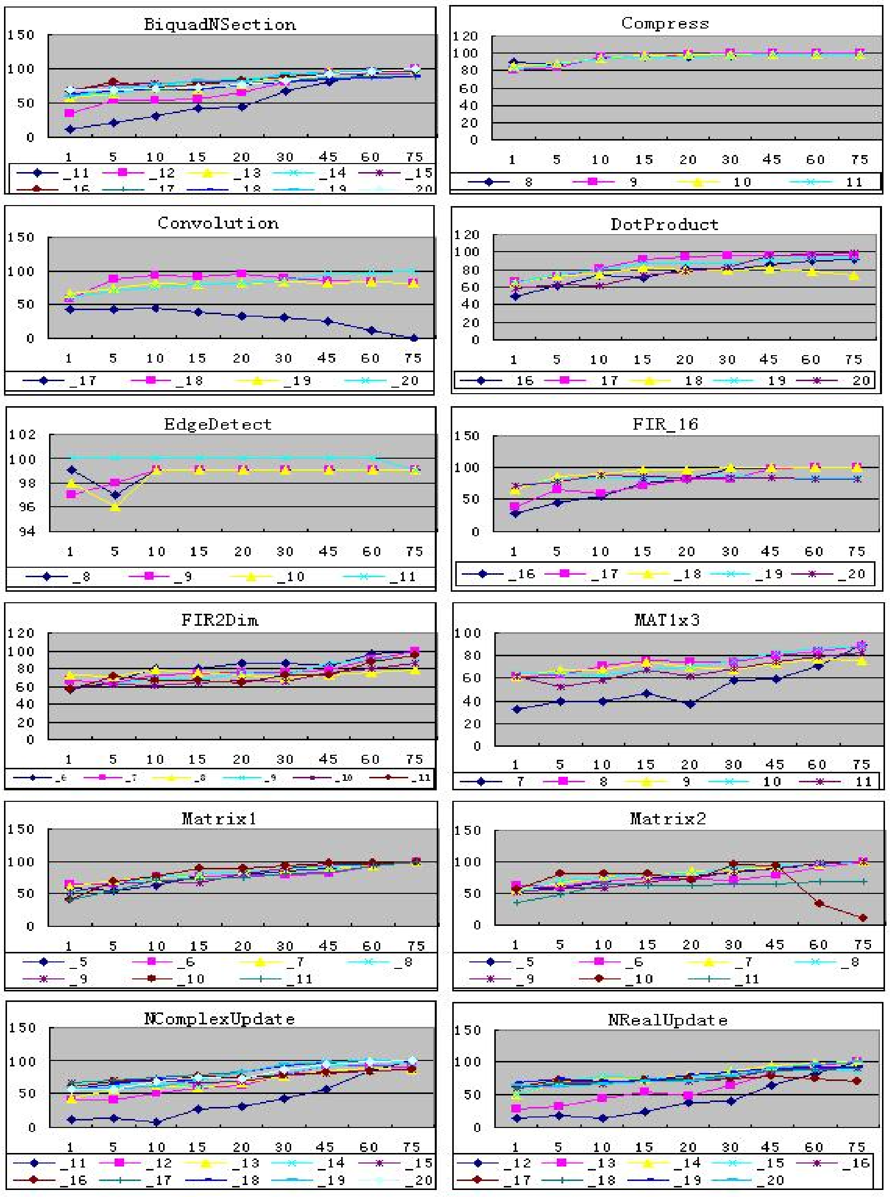

Fig. 2. The performance of PWA-IBL, where it is plotted against the size of training example set. The performance is defined as *r'/r*, where *r'* the average speedup achieved on a particular training set size *s* and *r* the highest speedup achieved on the given test case, regardless of training set size.

because it can learn from more examples and therefore select better parallel schemes.

However, there are some surges in the PWA-IBL curves on some test cases which deserve further analysis. These test cases include *Convolution_17*, *EdgeDetect_8*, *EdgeDetect_10* when the training set size *s*=5, *MAT1x3_7* when *s*=20, and *Matrix2_10* when *s*>45. In the case of *EdgeDetect_8* and *_10*, the surges are noise, as the actual variance is insignificant (between 96% and 98%), indicating that the training set selection has little impact on PWA-IBL performance. In the case of *MAT1x3_7*, a closer look at learning process shows that the training set contain many not-very-similar cases, which in turn affect the performance. Nevertheless, the highest speedup on *MAT1x3_7* is relatively modest at 1.74, with the variance less than 0.35.

Experimental results on *Convolution _17* show that speedup is achieved only with a scheme of 2^4=16 threads. No other scheme can improve its performance. Because the training cases are randomly chosen, when *s* increases, it becomes less likely that PWA-IBL can effectively predict this scheme, and its performance decreases instead.

In the case of *Matrix2_10*, higher speedups can be achieved as the number of threads increases. It reaches the highest when there are 2^{10}=1024 threads sharing the workload. A closer look at the raw experimental results show that besides *Matrix2_10* itself, only on *Matrix1_10*, *Matrix1_11* and *Matrix2_11* (all considered similar enough to *Matrix2_10*) similar speedup can be achieved with such a large number of threads. When the training set is small, the possibility is low for these programs to be included. PWA-IBL can only select a scheme of smaller number of threads, which results in a lower speedup. Therefore, the performance curve starts low. It starts rising when PWA-IBL is trained with a larger set of examples. When the size reaches a certain threshold (60 as indicated in Fig.2), it becomes more likely that one or more of these three programs be included in the training set. However, only on *Matrix1_10* the highest speedup is achieved with a scheme of 1024 threads, whilst on *Matrix1_11* and *Matrix2_11*, the highest speedup is achieved with 2^{11}=2048 threads. Therefore, PWA-IBL reaches the highest speedup only when it correctly selects *Matrix1_10* as the one most similar to *Matrix2_10*. Otherwise, no speedup is found since sharing the workload among 2048 threads degrades the performance. For *Matrix1_10* and *Matrix1_11*, sub-optimal speedups are achieved when PWA-IBL decides that 1024 is the best parallel scheme from its prior experience from *Matrix2_10* and *Matrix2_11*.

In brief, PWA-IBL finds the optimal parallel schemes for 36 out of the 76 test cases and over 80% of the highest speedups on the other 33. On average, its efficiency over these 76 test cases is 86%. It shows that PWA-IBL should be trained with at least 30 training examples to achieve a reasonable accuracy, In addition, whenever possible, training examples should be selected as diverse as possible to cover most of the also be able to adapt to such cases.

It is worth noting that the current experiment only considers test cases with both thread numbers and workloads proportional to 2, where the workload is evenly distributed among the threads. We leave cases with arbitrary workload and thread numbers for the future work, though preliminary results show that PWA-IBL should also be able to adapt to such cases.

5 Related Work

Parallelism [4][11] is one of the main sources for performance improvement in modern computing environments. Key techniques for automatic detection of parallelism are discussed in [5]. A prototype compiler is presented in [3], where loop transformation and parallelization techniques are used to achieve high performance on numerical Java codes. Jikes[2] compiler uses runtime feedback to direct its adaptive optimization. Dynamic optimization is presented in [21] to measure the cost of thread creation and parallelize code insides a Jikes RVM at run time. The dynamic parallelizing compiler in [17] uses runtime feedback to adapt both application and operating system to the JAMAICA chip multiprocessor (CMP) architecture on the fly. Speculative techniques[9][16] are developed to exploit parallelism[14][17]. Jrpm[8] is a CMP-based runtime parallelizing JVM with thread-level speculation support. Most of these work focus on the search of parallelism opportunity, load balancing and thread migration[14][21], and give little concern to the cost of parallelism and its impact on performance[18]. PWA-IBL can be considered a complement to them in helping the compiler to decide how the workload should be allocated at the first place. It can accelerate the search for best parallel scheme.

Machine learning[15] has recently been introduced to compiler optimization at system level. Various learning approaches are used in iterative optimization[1][10] to explore a large optimization space. Machine learning is used in [6] to build a performance model based on a small number of evaluations. It first tests a small set of sample optimizations on a prior set of benchmarks, then analyzes the results in order to identify characteristic optimizations, based on which some further test runs are carried out on the target program. This technique significantly reduces the cost of evaluating the impact of compiler optimizations. Logistic regression is used in [7] to derive a predictive model that selects suitable optimizations to apply to each method based on code features. Instance-based learning is used in [13] to select suitable loop transformations within an optimization space of various loop re-ordering transformations.

6 Conclusion

This paper presents a fast and efficient machine learning based parallel workload allocation approach. It uses static program features to classify programs, before deciding the best workload allocation based on its prior experience with similar programs. It can decide, in a cost-aware manner, the best number of threads to share the workload of a given loop in order to achieve optimal performance via multithreading. Experimental results show that PWA-IBL can find the best parallel schemes for 36 out of the 76 test cases and over 80% of the best speedups on the other 33. On average its efficiency over these 76 test cases is 86%. Its performance improves when trained with more examples.

We plan to use PWA-IBL to achieve runtime adaptability, where features such as the hardware counter reading could be used to estimate not only program but also system workload. Portability of a PWA-IBL-enabled compiler could be achieved via the introduction of some architectural features. In addition, PWA-IBL shall be enhanced for more general code block and arbitrary workload. Methods could be developed to select better training examples, identify less representative ones and eliminate them when necessary.

References

[1] F.Agakov, E.Bonilla, J.Cavazos, et.al, Using machine learning to focus iterative optimization, proc. of the 2006 International Symposium on Code Generation and Optimization, 2006.

[2] M.Arnold, M.Hind, B.Ryhder, Online feedback-directed Java optimization, ACM SIGPLAN Notices, 37(11), 2002.

[3] P.Artigas, M.Gupta, S.Midkiff,J.Moreira, Automatic loop transformation and parallelization for Java, proc. of the 14th International Conference. for Supercomputing, 2000.

[4] U.Banerjee, R.Eigenmann, A.Nicolau, D.Padua, Automatic program parallelization, Proceedings of the IEEE, 81(2), 1993.

[5] W.Blume, R.Eigenmann, J.Hoeflinger, et.al, Automatic detection of parallelism, a grand challenge for high performance computing, IEEE Parallel and Distributed Technology, 2(3), 1994.

[6] J.Cavazos, C.Dubach, F.Agakov, et.al, 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.

[7] J.Cavazos, M.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.

[8] M.Chen, K.Olukotun, The Jrpm System for Dynamically Parallelizing Java Programs, ACM SIGARCH Computer Architecture News, 31(2), 2003.

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

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

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

[12] M. Gupta, R. Nim. Techniques for speculative run-time parallelization of loops. proc. of Supercomputing '98, 1998.

[13] S.Long, M.O'Boyle, Adaptive Java optimization using instance-based learning, proc. of the 18th ACM International Conference on Supercomputing, France, 2004.

[14] P.Marcuello, A.Gonzales, J.Tubella, Speculative Multithreaded processors, proc. of the 1998 ACM International Conference on Supercomputing, 1998.

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

[16] J.Oplinger, D.Heine, M.Lam, In search of speculative thread-level parallelism, proc. of Parallel Architectures and Compilation Teniques (PACT'99). 1999

[17] G.Wright, A.El-Mahdy, I.Watson, Java machine and integrated circuit architecture, Java Microarchitecture, Kluwer, 2002

[18] L.Yang, J.Schopf, I.Foster, Conservative scheduling: using predicted variance to improve scheduling decisions in dynamic environments, proc. of Scientific Computing, 2003.

[19] J.Zhao, C.Kirkham, I.Rogers, Lazy interprocedural analysis for dynamic loop parallelization, proc. of Workshop on New Horizons in Compilers, India, 2006.

[20] J.Zhao, I.Rogers, C.Kirkham, I.Watson. Loop parallelization for the Jikes RVM, proc. of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2005), 2005.

[21] W.Zhu, C.L.Wang, C.M.Lau, JESSICA2: a distributed Java virtual machine with transparent thread migration support, proc. of IEEE 4th International Conference on Cluster Computing, 2002.

[22] V.Zivojnovic et.al, DSPstone: a DSP-oriented benchmarking methodology, proc. of Signal Processing Applications & Technology, 1994.