

Using Static Code Features for Intelligent Squash Prediction

Jeremy Singer, Adam Pocock, Paraskevas Yiapanis, Simon Wilkinson, Mikel Luján, and Gavin Brown
University of Manchester, UK

Email: {jsinger,apocock,pyiapanis,swilkinson,mlujan,gbrown}@cs.manchester.ac.uk

Abstract

Thread-level speculation (TLS) is widely accepted to be a realistic model for execution of sequential programs on multi-core architectures. One problem with TLS occurs when large numbers of spawned threads are squashed due to data dependence violations. This can reduce or entirely obliterate the performance benefits of TLS. Until now, informal compiler heuristics or high-overhead runtime profiling have been used to identify likely squashes and suppress thread spawns at the appropriate program locations. In this paper, we adopt an alternative approach using machine learning to construct squash predictors based on static code features. On a set of standard Java benchmarks, we achieve a mean speedup that is 82% of the maximum speedup available from an oracle predictor.

1. Introduction

It is a truth universally acknowledged that future improvement in program execution speed depends on increased *parallelism*. Moore’s law is now maintained by increasing the number of cores in a microprocessor, rather than increasing the complexity of an individual processor core. This implies that thread-level parallelism, rather than instruction-level parallelism, must be exposed in applications to generate speedups.

The primary motivation of *thread-level speculation* (TLS) is to enable parallel execution of sequential programs. This is achieved by selecting regions of a sequential program to execute in parallel threads. There is no absolute requirement that such threads should be parallelizable (indeed, if they definitely were, perhaps a programmer or compiler would have already applied parallelization optimization) so runtime mechanisms are required to detect and handle data dependence violations. TLS is most effective when the large majority of concurrently executing threads do not have cross-thread violations, since the cost of *squashing* specu-

lative threads and re-executing their code is generally high.

Thus the performance of TLS depends on solutions to the following two inter-dependent problems:

- 1) achieving effective parallelism from the sequential program (which relies on identifying good spawn points for speculative threads), and
- 2) avoiding speculative thread squashes (which relies on identifying potential conflicts ahead-of-time and preventing the speculative thread from spawning).

This paper uses speculative method-level parallelism (SMLP) to tackle the first challenge above. This is a standard TLS technique [1], [2], [3], [4]. Section 2 gives full details of our model of speculative execution.

However this paper concentrates on addressing the second challenge above, in a more formal way than has been considered previously. Until now, squash prediction has been based on human-generated heuristics, derived by experts with domain-specific knowledge [5]. Our research aims to provide heuristics with a more rigorous derivation, using *machine learning* techniques.

We now present a simple motivating example. Suppose method m_1 has a set of properties P_1 , and method m_2 has a set of properties P_2 . Whenever they are speculatively executed in parallel, there is a data dependence violation. From this information, we may construct a rule that says something like: ‘never spawn methods that have property sets P_1 in parallel with methods that have property sets P_2 .’ Thus we have *generalized* from the specific case for m_1 and m_2 to arbitrary methods in any programs. Section 3 specifies precisely the actual method-level properties that we measure. These properties are known as *nano-patterns*. For instance, P_1 may be ‘{ reads values from array, contains loop }’ and P_2 may be ‘{ creates array, writes values to array }.’ One can see how this might cause problems with method-level speculation in the (not untypical) code shown in Figure 1. If the

```

void m1() {
    int a[];
    int sum = 0;
    // — THREAD SPAWN POINT —
    m2(a);

    // — CONTINUATION of m1 —
    for (int i=0; i<a.length; i++) {
        sum = sum + a[i];
    }
    output(sum);
}

void m2(int a[]) {
    // create array
    a = new int[10];

    // init non-0 array elements
    a[1] = 42;
    a[5] = 17;
    a[9] = -1;
}

```

Figure 1. Example Java code to demonstrate squashes in speculative method-level parallelism

body of method m_2 is executed in parallel with the continuation of m_1 , there will be data dependence violations due to conflicting accesses to elements of array a .

A simple data flow analysis would determine that m_2 must write values that m_1 later reads, in this particular case. However in general the situation is complicated by aliased pointers, irregular control flow and other analysis complexities.

The beauty of speculative execution is that it allows our prediction algorithms to make occasional mistakes. We can afford intermittent squashes, so long as their overhead does not negate the benefits of parallelization. Therefore we decide to investigate squash prediction based on simple static properties of methods. Whereas complex static or dynamic analysis may be more accurate, they are also more expensive. In the context of Java just-in-time compilation, these heavyweight techniques are not appropriate since the overhead of runtime compilation is exposed to the end-user.

We use machine learning to relate static features of Java methods to squashing behaviour. Section 4 outlines our approach. Since the features are general-purpose, we can train our predictor ahead-of-time on a standard set of Java programs, then test our predictor on a previously unseen program. Thus, as far as the end-user is concerned, there is no application execution time hijacked for program profiling.

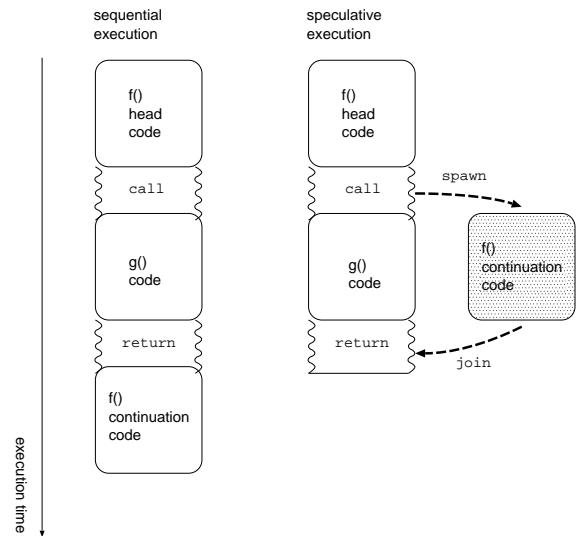


Figure 2. Speculative execution model

This paper makes the following key contributions:

- 1) It demonstrates the utility of static method-level properties for squash prediction on previously unseen code, using machine learning techniques.
- 2) It shows that an oracle squash predictor can improve the performance of speculative method-level parallelism by up to 64% over sequential execution.
- 3) It shows that an intelligent squash predictor, generated by the CPAR algorithm, is able to achieve 82% of the performance improvement due to an oracle predictor.

2. Speculation Model

2.1. Speculative Method-Level Parallelism

Since the Java programming language is *object-oriented*, the natural unit of abstract behaviour is the *method*. Thus we assume that distinct methods are likely to have independent behaviour, so methods are suitable code segments for scheduling as parallel threads of execution.

Figure 2 presents a graphical overview of how SMLP operates, given a method f that calls a method g . A speculative thread is spawned at the method call to g . The original non-speculative thread continues to execute the body of g , without any change in its speculative status. The new thread skips over the method call and starts execution at the point where g returns to the continuation of f . This new child thread

is in a more speculative state than its parent spawner thread.

During the subsequent parallel execution of these two threads, if the parent writes to a memory location that has been read by the child, then we have a *data dependence violation*. The child speculation must be squashed, and the method continuation re-executed. On the other hand, if the parent thread completes the method call without causing any data dependence violations, then the spawnee can be committed. This means that its speculative actions can be confirmed to the whole system, and the spawnee is joined to the spawner. Execution resumes from where the spawnee was at the join point, in the less speculative state of the spawner.

The SMLP model permits *in-order* nested speculation, which means that spawned threads can in turn spawn further threads at more speculative levels. However if a spawner thread itself has to be squashed, all its spawned threads must also be squashed.

Note that there are overheads for spawning new threads, committing speculative threads and squashing mis-speculations. Speculation must be carefully controlled to avoid excessive mis-speculation and the corresponding performance penalty. This motivates our interest in accurate *squash prediction* techniques.

In our architectural model, we make a number of simplifying assumptions, justified since we are engaged in a limits study for the impact of squash prediction on SMLP performance.

Write buffering: A speculative thread keeps its memory write actions private until the speculation is committed. This requires buffering of speculative state until the commit event. We assume buffers have *infinite size*. **Return value prediction:** If a method continuation (executing as a speculative thread) depends on the value of a method call (executing concurrently as a less speculative thread), then we assume that the return value may be predicted with *perfect accuracy*. **Data forwarding:** We do not allow a less speculative thread to forward its write values to a more speculative thread, after the more speculative thread has been spawned, i.e. there is *no data forwarding*. **Input/output:** We assume that all I/O interaction in speculative threads is completely buffered. **Parallelism:** All benchmark execution takes place with an *infinite number of cores* available for speculative multi-threading.

2.2. Squash Prediction

Once we have fixed our TLS model so that spawns are only allowed to occur at method calls, the next task is to determine which potential spawn points

we should ignore. Obviously we could spawn a new speculative thread at each method call; however many of these spawns will not lead to performance gain, either because the spawned method is too short for the parallelism to outweigh the overhead of thread creation, or because the spawned thread is guaranteed to cause a data dependence violation resulting in a squash.

It is sometimes possible to eliminate certain useless spawn points via static analysis. Other spawns are eliminated as a result of dynamic profiling that studies their behaviour over a program run. In general, proposed TLS systems employ either or both of these techniques to eliminate poor spawn points. We aim to create a *hybrid* scheme, that can generate advice about spawn points based on features of program code (like *static* analysis) given prior knowledge of runtime behaviour of that same, or similar, code (like *dynamic* analysis).

The relationship between static code features and dynamic squashing behaviour is constructed using *machine learning* algorithms. Thus we are able to create general *squash predictors* that can provide advice for any code, whether previously seen or unseen. This is a key strength of learning-based techniques, which has not been exploited previously in the TLS domain.

Note that use of learning-based predictors does not prevent further runtime profiling to fine-tune spawn point advice dynamically. At present, we envisage offline learning for ahead-of-time predictions, as a drop-in replacement for static spawn point elimination. Previously such static elimination was based on ad-hoc compiler heuristics, whereas now we are proposing to apply well-understood learning techniques to enable more intelligent squash prediction.

2.3. Simulation

All speculative execution is simulated using a trace-driven simulation methodology. We run sequential, single-threaded Java benchmark applications from the DaCapo suite [6] on Jikes RVM v2.9.3 [7] within the Simics v3.0.31 full-system simulation environment [8]. Simics is configured for IA-32 Linux, using the supplied *tango* machine description. We use a *perfect memory* model, justified since this is a limits study. Jikes RVM is instrumented to enable call-backs into Simics to record significant runtime events. These include method entry and exit, heap read and write, exception throw, etc. Each event is recorded with appropriate metadata, such as method identifier, memory address, and processor cycle count.

Thus we produce a sequential execution trace of events that may affect speculative execution. We feed the trace file to a custom TLS simulator. It uses method call information to drive speculative thread spawns, and heap memory access information to drive thread squashes based on data dependence violations. The timing information in the sequential trace enables the TLS simulator to determine method runlengths, in order to estimate an execution time based on parallel execution once it has determined which spawned threads commit or squash.

Note that the sequential trace is pre-processed to discover methods with runlengths shorter than a fixed threshold. The TLS simulator is configured never to spawn on these calls, since the overhead of spawning outweighs the benefit of parallel execution of a short method [2]. We use a threshold of 1000 cycles throughout this paper. Again following earlier work [2], [9] we have fixed costs for thread spawn, commit and squash events. These costs are parameterizable in our custom TLS simulator.

Finally, we emphasize that all performance improvements in our system are due to *execution time overlap*. We do not model any secondary effects due to warming up caches and other architectural units. Other researchers quantify the benefit of this helper-thread effect of speculative execution, and find it to be a large component of the overall performance improvement [10].

3. Nano-Patterns

Fundamental nano-patterns are simple, static, binary properties exhibited by Java methods. They are *traceable*; that is, ‘they can be expressed as a simple formal condition on the attributes, types, name and body’ of a Java method [11]. They should be automatically recognisable by a trivial static analysis of Java bytecode.

Nano-patterns were originally proposed by Gil and Maman [11], however they did not develop the idea significantly. Instead, Gil and Maman focus on *micro patterns* which are properties of Java classes. Høst and Østvold [12] present the first catalogue of *traceable attributes* for Java methods. They argue that these attributes could be used as building blocks for defining nano-patterns. In this paper, we refer to such attributes as *fundamental nano-patterns*, which could potentially be combined to make *composite nano patterns*.

We have supplemented the initial catalogue from Høst and Østvold, by incorporating further attributes that we intuitively feel should be relevant for speculative method-level parallelism. These include fundamental nano-patterns that capture method calling

relationships and array accesses. We have created a command line tool to detect these nano-patterns for methods in Java bytecode class files, based on the ASM bytecode analysis toolkit [13]. The current set of 19 fundamental nano-patterns that we track is given in Figure 3.

Conceivably, nano-pattern information could be encoded as annotations in class files at source to bytecode compilation time. Instead we simply opt to generate a separate table of nano-patterns, indexed by the concatenation of a fully qualified class name with a method signature. Our database contains all methods present in bytecode from our set of benchmark applications, the standard class library and the Jikes RVM runtime environment.

Nano-patterns provide *concise* and *abstract* summaries of Java method characteristics. Therefore they are ideal as *features* for machine learning algorithms.

4. Learning Problem

We employ *supervised* machine learning, which means that we provide a set of examples; each example is characterized by *feature* values and labelled with a *class*. When the learning algorithm is given such a training set, it generates a *classifier* that describes each class in terms of its feature values. Different learning algorithms produce different kinds of classifiers. In our case, we use a kind of *rule induction*, described fully in Section 5.2.

The classifier can be tested by providing *unlabelled* examples, characterized by the same set of features as the training set. The classifier will predict the class for each example. We can determine the classifier accuracy by calculating the actual class for each example and comparing this with the predicted class.

The particular learning problem we handle is *squash prediction* at thread spawn points in SMLP. Recall from Section 2 that at a spawn point, we have two interacting methods: *caller* and *callee*. We characterize each method by a set of 19 nano-pattern feature values, as introduced in Section 3.

We need to generate *training data* that has labels, i.e. we must determine at each spawn point whether or not a squash occurs if a speculative thread is spawned there. We run training programs and issue a spawn at every call site where the callee method runlength is above a certain threshold [2]. The trace-based simulator can detect squashes due to data dependence violations. Thus we can classify each dynamic spawn point with a commit / squash class label. So each example in our training data set has the form (19 binary nano-pattern values for caller method) followed by (19

<i>category</i>	<i>name (abbrev)</i>	<i>description</i>
Calling	NoParams (N) NoReturn (V) Recursive (R) SameName (S) Leaf (L)	takes no arguments returns <code>void</code> calls itself recursively calls another method with the same name does not issue any method calls
Object-Orientation	ObjectCreator (OC) InstanceFieldReader (IFR) InstanceFieldWriter (IFW) StaticFieldReader (SFR) StaticFieldWriter (SFW) TypeManipulator (TM)	creates <code>new</code> objects reads field values from an object writes values to field of an object reads static field values from a class writes values to static field of a class uses type casts or <code>instanceof</code> operations
Control Flow	StraightLine (SL) Looping (LO) Exceptions (E)	no branches in method body one or more control flow loops in method body may throw an unhandled exception
Data Flow	LocalReader (LR) LocalWriter (LW) ArrayCreator (AC) ArrayReader (AR) ArrayWriter (AW)	reads values of local variables on stack frame writes values of local variables on stack frame creates a new array reads values from an array writes values to an array

Figure 3. Catalogue of fundamental nano-patterns. Boldface names denote original patterns we have devised, all other patterns are due to Høst and Østvold.

<i>Benchmark</i>	<i># spawns</i>	<i># squashes</i>	<i>% squashes</i>
antlr	169578	38049	22.4%
bloat	1282628	93680	7.3%
fop	278712	54780	19.7%
lusearch	811	133	16.4%
pmd	39046	18815	48.2%

Figure 4. Statistics for per-benchmark training data set

binary nano-pattern values for callee method) followed by (squash OR commit).

We extract this training data from five Java benchmarks from the DaCapo suite, running with `small` input data sets. Figure 4 presents some statistics for each of these benchmarks.

5. Evaluation

Our evaluation is split into three parts. Section 5.1 analyses the correlations between nano-patterns and squash behaviour; Section 5.2 examines the rules that our learning algorithm generates; and Section 5.3 reports the SMLP performance improvements with squash prediction.

5.1. Information Theoretic Feature Analysis

Information theory provides a framework for analysing the information contained in variables and the correlation between variables in a non-parametric way. The mutual information provides a value for the amount of information that is contained in one variable

about another, which in the binary case is in the range $[0, 1]$. The reliability of mutual information calculations depends on the dimensionality of the feature set.

mRMR (minimum Redundancy, Maximum Relevance) is an information theoretic algorithm to select a group of features that capture all the information about the class [14] whilst avoiding the dimensional problems caused by using the mutual information. mRMR can be used to order the features by their total relevance to the class.

The mRMR algorithm selects features which are both highly informative and do not contain information already held in the selected feature set. The information measure used to select the features turns negative when the addition of a new feature does not add more information to the selected feature set. This can be used to provide a limit on the size of the selected feature set.

Each benchmark data set contains at most 11 features that are jointly relevant to the squash behaviour. We also consider the combined data set with all benchmark data.

Figure 5 shows the nano-patterns which were selected more than once, ordered by the frequency of selection across all the data sets. Callee method nano-patterns have a ' suffix. Many of these features are interpretable, given some domain knowledge. For instance, callee leaf methods are good indicators for squash predictors. Leaf methods are likely to have few side-effects, and therefore unable to cause many data dependence violations in general.

On the other hand, there are three callee nano-patterns involving heap memory writes. These are

<i>nano-pattern</i>	<i>frequency</i>
LR	6
L'	6
LO'	5
IFW'	5
OC'	4
AW'	4
LR'	4
E	3
TM	3
R	3
SFW'	3
TM'	2
OC	2
IFW	2
V	2
E'	2
N	2
LO	2
SL'	2

Figure 5. Features selected by the mRMR algorithm, ordered by frequency (number of data sets for which this feature is found to be relevant)

all prominent in the list, since heap memory write accesses in the callee method are prime causes of data dependence violations, if the caller continuation reads from the written location during its speculative execution.

5.2. Associative Classification

Associative Classification (AC) [15], [16], [17] uses frequently occurring patterns to build a set of rules that form a *classifier*. The generated rules are Class Association Rules (CARs) which have the particular form $A \rightarrow B$. The antecedent A of the rule can contain any number of attribute-value pairs known as items. The consequent B is restricted to only one item which is the class label (i.e. the predicted class). A and B are disjoint items. A rule is also accompanied with two measures to express its interestingness. The *support* is simply the proportion of examples that exhibit both A and B in relation to the entire dataset. The *confidence* (or accuracy) is the proportion of examples that exhibit both items A and B in relation to the total number of examples that exhibit A . A CAR is only retained if it satisfies user-specified minimum thresholds for both support and confidence [18]. In all our experiments, we set the confidence threshold for squash prediction rules to be 80%. We do not specify a high support

threshold since the data sets are so large and diverse.

In our experiments we have applied an AC algorithm known as CPAR (Classification based on Predictive Association Rules) [15]. Since association rules only admit positive features (in our case: nano-pattern presence, not absence, in a method) we double our feature vector by including the logical complement of each nano-pattern as a separate feature. We have seen from earlier investigations that this improves the performance of the generated classifier.

The algorithm takes two input datasets: The training set is used to generate the rules, and the test set is used to evaluate the predictor. In our experiments, we use Leave-one-out Cross-validation (LOOCV). For n benchmark programs, data from $n - 1$ benchmarks are merged to form the training set, and the one left out is used as the test set. The same idea is applied until all the benchmarks have been used for test sets.

Below is an example rule generated by the CPAR algorithm, that has a confidence of 87% and a support of 1.4%.

$$V \& LW \& TM' \& LO' \rightarrow \text{squash}$$

We note that three of these four features are marked by the feature selection algorithm in Figure 5. We can envisage that a looping callee might be a long-running method, and therefore it might have more potential for causing a data dependence violation.

On average, each of the LOOCV training data sets produced 128 CPAR rules. The classification algorithm took up to 30 minutes to complete, on a 3GHz IA-32 workstation with 2GB RAM.

5.3. Performance

We evaluate SMLP execution on our TLS simulator. Since this is a limits study, we use contrived TLS overhead costs to emphasize the performance improvement that is possible with accurate squash prediction. Thus we set the cost of thread spawn and commit to 0 cycles, and the cost of thread squash to 500 cycles. These are the limits of the costs in the TLS model evaluated by Warg and Stenstrom [2].

We compare benchmark performance with various squash prediction strategies, outlined below:

- 1) **squash always**: equivalent to sequential computation. We take this as the baseline speed of 1.0 in Figure 6.
- 2) **commit always**: always spawn speculative threads for methods that have a runlength above the threshold.
- 3) **random**: 50/50 coin-flip predictor for squash/commit. Any machine learning predictor must

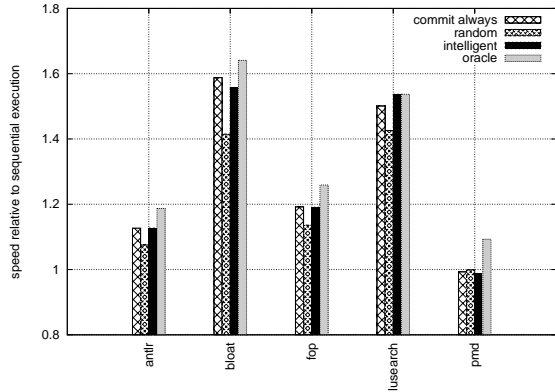


Figure 6. Relative SMLP performance of various squash prediction schemes on Java benchmarks

be able to outperform this.

- 4) **intelligent**: Use a rules-based predictor for squashes.
- 5) **oracle**: 100% accurate predictor, only (and always) allows spawns for guaranteed commits. This means that squashes never occur, which gives the hypothetical limit on performance of any real predictor.

Figure 6 shows the speedups over sequential execution that can be obtained by various squash prediction strategies. In 4 out of 5 cases, the **intelligent** predictor beats the **random** predictor. In the case of **pmd**, no spawning strategy even achieves the baseline sequential speed, due to the high proportion of squashes in this benchmark. In 3 cases, the **intelligent** predictor is as good as or better than the **commit always** predictor. In the case of **lusearch**, the **intelligent** predictor gives comparable performance to the **oracle** predictor.

On average, the **intelligent** predictor gives 82% of the speedup that is attained by the **oracle** predictor.

6. Related Work

In earlier work [19] we have used fundamental nano-patterns to classify and cluster Java benchmark programs based on their object-oriented properties. To the best of our knowledge, nano-patterns have not previously been used in the context of SMLP.

Whaley [9] presents a set of seven heuristics for controlling spawns in SMLP execution. His heuristics are manually generated from expert domain knowledge. The heuristics require dynamic information such as method runlengths and profile information. Thus programs require ahead-of-time profiling before heuristics can be applied. Our machine-learning based approach

has the advantage that it can train on a set of programs, then be applied to a previously unseen program. Whaley gives oracle predictor speedup figures of around 1.5 on a set of Java benchmarks. His heuristics are able to achieve around 80% of this optimal speedup.

The POSH TLS compiler [10] uses program structure (loops and method calls) to identify potential spawn points. These spawn points are then *refined* by means of runtime profiling to eliminate useless spawns. The authors report a mean 1.3 speedup on standard benchmarks. They attribute 26% of the speedup to prefetching effects, which we do not consider.

Other researchers use similar profile feedback information to drive the compiler’s decisions on spawn point insertion [20], [21]. In contrast, Dou and Cintra [5] present an entirely static cost model of TLS, which allows the compiler to estimate speedups in loop-level speculation with fairly high accuracy, and thus refine spawn insertions. They claim the framework gives a 25% speedup over a naive ‘spawn everywhere’ approach. There are also entirely *dynamic* squash prediction techniques. For instance, Cintra and Torrellas [22] present a hardware lookup table approach that learns which data dependences are likely to cause thread squashes as the TLS program executes. Thus it is able to adapt dynamically the spawning policy for threads that eliminate many squashes. Our machine-learning based approach would complement such a runtime technique.

Warg and Stenstrom [2] present another heuristic approach to improving SMLP. They observe that short methods should not be parallelized since the speculative overhead outweighs the benefit of parallel execution. They use a dynamic runlength predictor to identify short methods, and suppress spawns at these method calls. They use several different method runlength thresholds for spawn suppression, varying between 0 and 500 cycles. We adopt a similar policy in all our experiments, assuming perfect runlength prediction and a threshold of 1000 cycles.

7. Conclusions

In summary, we have presented a machine learning based approach to squash prediction, using static code features of methods. Our results show that the predictors can achieve 82% of the optimal speedup for SMLP on a set of standard Java benchmarks. However we feel that we have not harnessed the full potential of machine learning. After all, the rules-based **intelligent** predictor is only just outperforming the simple **commit always** predictor. There are problems due to the *imbalance* between squash and commit

frequencies in the benchmark data sets. In addition, the data sets have vastly *differing sizes*. Our simple rules-based learning strategy shows some promise, but we can better if we can handle these data set problems using more sophisticated learning techniques.

With regard to other future work, we hope to evaluate further static features for squash prediction. We also aim to incorporate dynamic information (along the lines of Whaley [9]) into our learning system since we anticipate that this should further improve the predictor accuracy.

References

- [1] M. K. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded Java programs," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 176–184.
- [2] F. Warg and P. Stenström, "Improving speculative thread-level parallelism through module run-length prediction," in *Proc. International Symposium on Parallel and Distributed Processing*, 2003, p. 12b.
- [3] J. Oplinger, D. Heine, and M. Lam, "In search of speculative thread-level parallelism," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, 1999, pp. 303–313.
- [4] C. J. F. Pickett and C. Verbrugge, "Software thread level speculation for the Java language and virtual machine environment," in *Proc. Workshop on Languages and Compilers for Parallel Computing*, 2005, pp. 304–318.
- [5] J. Dou and M. Cintra, "A compiler cost model for speculative parallelization," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 2, p. 12, 2007.
- [6] S. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.
- [7] B. Alpern *et al.*, "The Jalapeno virtual machine," *IBM Systems Journal*, vol. 39, no. 1, p. 211, 2000.
- [8] P. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.
- [9] J. Whaley and C. Kozyrakis, "Heuristics for profile-driven method-level speculative parallelization," in *Proc. 34th International Conference on Parallel Processing*, 2005, pp. 147–156.
- [10] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: a TLS compiler that exploits program structure," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006, pp. 158–167.
- [11] Y. Gil and I. Maman, "Micro patterns in Java code," in *OOPSLA*, 2005, pp. 97–116.
- [12] E. W. Høst and B. M. Østvold, "The programmer's lexicon, volume I: The verbs," in *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 193–202.
- [13] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," in *Adaptable and Extensible Component Systems*, 2002.
- [14] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, 2005.
- [15] X. Yin and J. Han, "CPAR: Classification based on predictive association rules," in *Proc. SIAM International Conference on Data Mining*, 2003, pp. 369–376.
- [16] W. Li, J. Han, and J. Pei, "CMAR: Accurate and efficient classification based on multiple class-association rules," in *Proc. International Conference on Data Mining*, 2001, pp. 369–376.
- [17] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *Proc. KDD*, 1998, pp. 80–86.
- [18] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. International Conference on Very Large Databases*, 1994, pp. 487–499.
- [19] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yianpanis, "Fundamental nano-patterns to characterize and classify Java methods," in *Proc. Workshop on Language Descriptions, Tools and Applications*, 2009, (to appear).
- [20] P. Wu, A. Kejariwal, and C. Cascaval, "Compiler-driven dependence profiling to guide program parallelization," in *Proc. Workshop on Languages and Compilers for Parallel Computing*, 2008, pp. 232–248.
- [21] M. K. Chen and K. Olukotun, "The Jrpm system for dynamically parallelizing Java programs," in *Proc. International Symposium on Computer Architecture*, 2003, pp. 434–446.
- [22] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors," in *Proc. International Symposium on High-Performance Computer Architecture*, 2002, p. 43.