Static Java Program Features for Intelligent Squash Prediction

Jeremy Singer, Paraskevas Yiapanis, Adam Pocock, Mikel Lujan, Gavin Brown, Nikolas Ioannou, and Marcelo Cintra

[1] University of Manchester, UK
jsinger@cs.man.ac.uk
[2] University of Edinburgh, UK

Abstract. The *thread-level speculation* paradigm parallelizes sequential applications at run-time, via optimistic execution of potentially independent threads. This enables unmodified sequential applications to exploit thread-level parallelism on modern multicore architectures. However a high frequency of data dependence violations between speculative threads can severely degrade the performance of thread-level speculation. Thus it is crucial to be able to schedule speculations to avoid excessive data dependence violations. Previous work in this area relies mainly on program profiling or simple heuristics to avoid thread squashes. In this paper, we investigate the use of *machine learning* to construct squash predictors based on static program features. On a set of standard Java benchmarks, with leave-one-out cross-validation, our approach significantly improves speculation performance for two benchmarks, but unfortunately degrades it for another two, in relation to a spawn-everywhere policy. We discuss how to advance research on squash prediction, directed by machine learning.

1 Introduction

With the emergence of multi-core architectures, it is inevitable that parallel programs are favored as they are able to take advantage of the available computing resource. However there is a huge amount of legacy sequential code. Additionally, parallel programs are difficult to write as they require advanced programming skills. Some state-of-the-art compilers can automatically parallelize sequential code in order to run on a multi-core system. However such compilers conservatively refuse to parallelize code where data dependencies are ambiguous. Thread-Level Speculation (TLS) has received a lot of attention in recent years as a means of facilitating aggressive auto-parallelization [1] [2] [3] [4] [5] [6] [7] [8]. TLS neglects any ambiguities in terms of dependencies and proceeds in parallel with future computation in a separate speculative state as if those dependencies were absent. The results are then checked for correctness. If they are correct, the speculative state can safely write its side effects back to memory (*i.e.* commit). If the results are wrong, all speculative state is discarded and the computation is re-executed serially. (*i.e.* squash).

A high number of squashes results in *performance degradation* as:

1. There is a relatively high overhead associated with thread-management (roll-back and re-execute).
2. Squashed threads waste processor cycles that could usefully be allocated to other non-violating parallel threads.

An optimal situation would be one that no cross-thread violation occurs and therefore all speculative threads can commit their state. The spawning policy (the speculation level) employed by a TLS system is an important factor here. However spawning policy alone cannot guarantee the absence of data dependence violations. Ideally we would like to have a mechanism that can detect conflicts ahead-of-time and thus ultimately decide whether to spawn a thread or not. In this paper we simulate method-level speculation or *Speculative Method-level Parallelism (SMLP)* in order to collect data about speculative threads that commit or squash. We then mine static characteristics of these Java methods using Machine Learning in order to relate general method properties to TLS behavior.

The main contributions of this paper are:

- a description of static program characteristics that may provide useful features for learning about Java methods (Section 3).
- a comparative evaluation of profile-based and learning-based policies for squash prediction, in the context of method-level speculation for Java programs (Section 4).
- an outline of future directions for investigation into learning-based squash prediction (Section 6).

2 Speculation Model

2.1 Speculative Method-Level Parallelism

Since the Java programming language is *object-oriented*, the natural unit of abstract behavior is the *method*. Thus we assume that distinct methods are likely to have independent behavior, so methods are suitable code segments for scheduling as parallel threads of execution [9] [10] [11] [12].

Figure 1 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

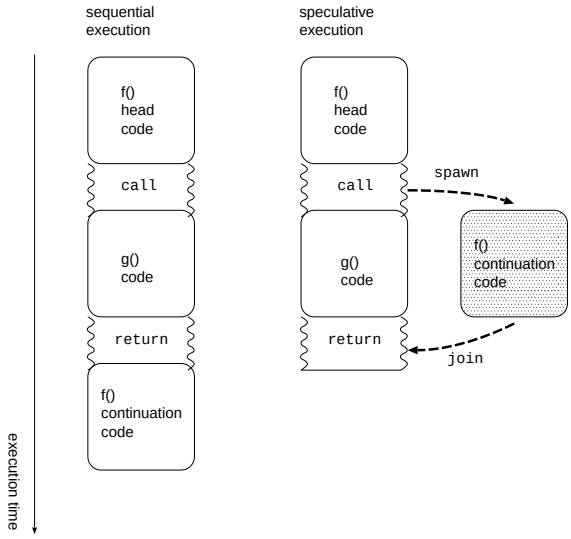

Fig. 1. Speculative execution model

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 two idealized assumptions:

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*.

2.2 Benchmarks

This investigation uses Java applications from the SpecJVM98 [13] and DaCapo [14] benchmark suites, which are widely used in the research domain. Programs from different suites and genres are helpful to assess how our predictions general-

ize, for previously unseen data. An overview of the selected benchmarks is shown in Figure 2. We use `s1` inputs for SpecJVM98 and `small` inputs for DaCapo.

benchmark	*description*
_202_jess	AI problem solver
_205_raytrace	raytracing graphics
_213_javac	Java compiler
_222_mpegaudio	audio decoding
_228_jack	parser generator
antlr	parser generator
fop	PDF graphics renderer
pmd	Java bytecode analyser

Fig. 2. Benchmarks.

2.3 Trace-driven Simulation

All speculative execution is simulated using *trace-driven simulation*. Each sequential, single-threaded Java benchmark application executes with Jikes RVM v2.9.3 [15] in the Simics v3.0.31 full-system simulation environment [16]. Simics is configured for IA-32 Linux, using the supplied `tango` machine description, with a *perfect memory* model. The Jikes RVM compiler 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.

For the sake of simplicity, the TLS simulator only has two processors. Only two methods can execute in parallel. Methods are considered as candidates for spawning if their total sequential runlength is between 1000 and 10,000 cycles. If both available cores are occupied, then new speculations cannot be scheduled, i.e. the speculation is *in-order*. We impose a small, fixed 10 cycle overhead for each TLS spawn, squash and commit event during program execution.

All performance improvements in our simulated TLS 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 [5] [17].

3 Intelligent Squash Prediction in TLS

3.1 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 behavior 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 behavior of that same, or similar, code (like *dynamic* analysis).

The relationship between static code features and dynamic squashing behavior 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.

3.2 Feature Collection

This study focuses entirely on *static* features, i.e. information about a program that can be gained from an inspection of its source or object code, without actually executing the program. We characterize each Java method by 45 features. 22 features are *fundamental nano-patterns*, described in [18] and [19]. These are binary properties of Java methods that can be easily extracted by trivial static analysis of the bytecode. The patterns denote a summary of the behavior and characteristics a method exhibits (such as array accesses and method calling relationships). Figure 3 outlines the patterns we collect.

We collect a further 23 integer-valued characteristics for each method, derived from the MILEPOST feature set [20]. These measurements are taken on the

Feature	Meaning
NoParams (N)	takes no arguments
NoReturn (V)	returns `void`
Recursive (R)	calls itself recursively
SameName (S)	calls another method with the same name
AbstractCaller (A)	issues calls via `abstract` methods
Leaf (L)	does not issue any method calls
ObjectCreator (OC)	creates `new` objects
ThisInstanceFieldReader (TFR)	reads field values from `this` object
ThisInstanceFieldWriter (TFW)	writes values to field of `this` object
OtherInstanceFieldReader (IFR)	reads field values from some object
OtherInstanceFieldWriter (IFW)	writes values to field of some object
StaticFieldReader (SFR)	reads static field values from a class
StaticFieldWriter (SFW)	writes values to static field of a class
TypeManipulator (TM)	uses type casts or instanceof operations
StraightLine (SL)	no branches in method body
Looping (LO)	one or more control flow loops in method body
Exceptions (E)	may throw an unhandled exception
LocalReader (LR)	reads values of local variables on stack frame
LocalWriter (LW)	writes values of local variables on stack frame
ArrayCreator (AC)	creates a new array
ArrayReader (AR)	reads values from an array
ArrayWriter (AW)	writes values to an array

Fig. 3. Java method-level features, based on fundamental nano-patterns

compiler intermediate form (in our case, Jikes RVM HIR code). They relate to the size and shape of the control flow graph, and the mixture of instruction kinds. Figure 4 outlines these features.

3.3 Learning Problem

We treat each method call in a dynamic execution trace as a potential TLS spawn point. At each spawn, two methods are involved: the caller and the callee. We characterize the spawn by a vector of 90 features: 45 from each method involved. The class to predict is a binary summary of the outcome of this speculation: commit or squash. We employ *supervised learning*, which means that we have a set of *labeled* samples to create the classification model. We use the C5.0 [21] algorithm to construct a rules-based classifier.

4 Evaluation

We use *leave-one-out cross-validation* (LOOCV) to evaluate our squash prediction classifiers. When we want to evaluate the squash predictor for benchmark b, we gather training data from the other benchmarks (excluding b) to generate

Feature	Meaning
bbNum	Number of basic blocks
bbOneSuc	Number of basic blocks with one successor
bbTwoSuc	Number of basic blocks with two successors
bbMtTwoSuc	Number of basic blocks with more than two successors
bbOnePred	Number of basic blocks with a single predecessor
bbTwoPred	Number of basic blocks with two predecessors
bbMtTwoPred	Number of basic blocks with more than two predecessors
bbOnePredOneSuc	Number of basic blocks with a single predecessor and a single successor
bbOnePredTwoSuc	Number of basic blocks with a single predecessor and two successors
bbTwoPredOneSuc	Number of basic blocks with two predecessors and a single successor
bbTwoPredTwoSuc	Number of basic blocks with two predecessors and two successors
bbMtTwoPredMtTwoSuc	Number of basic blocks with more than two predecessors and more than two successors
bbInstrLt15	Number of basic blocks with number of instructions less than 15
bbInstr15and500	Number of basic blocks with number of instructions in the interval [15,500]
bbInstrMt500	Number of basic blocks with number of instructions greater than 500
directCallNum	Number of direct calls in the method
cndBrnchNum	Number of conditional branches in the method
uncndBrnchNum	Number of unconditional branches in the method
methodInstrNum	Number of instructions in the method
bbInstrNum	Average number of instructions in basic blocks
loadNum	Number of memory load instructions in the method
storeNum	Number of memory store instructions in the method
allocNum	Number of memory-allocating instructions in the method

Fig. 4. Java method-level features obtained from the compiler intermediate form, based on the MILEPOST feature set

the classifier, which will then be applied to b. In this way, we test the generality of the squash prediction rules since they are always applied to previously unseen data.

The C5.0 algorithm generates a set of rules for predicting squashes and commits. Each rule has an associated *confidence* based on its accuracy in the training set. We select rules that predict squashes with a confidence above 90%, to apply in our testing rule set. Rules are incorporated directly into our TLS simulator via C header files. At each method call in the simulated execution, the rules are applied to check method properties and predict whether this potential spawn would result in a squash. If no squash is predicted and a free core is available, then the TLS spawn event occurs in the simulator. In an actual TLS system, such static rules would probably be encoded as static hints at compile time. A small amount of runtime support might be required for some dynamically dispatched methods.

Figure 5 compares three squash prediction policies for method-level speculation on a 2-core TLS system. For each benchmark, the left-most bar shows the speedup (over sequential execution) for no squash prediction. The middle bar

shows the speedup for profile-based prediction, which involves determining the overall proportion of squashes at each callsite in the program, during a profiling run. Then during the test run, squashes are predicted for call sites that had more than 90% squashes on the profiling run. This is not machine learning—it is simple statistical analysis, training and testing on the same data. Finally, the right bar shows the speedup with learning-based prediction, using the squash prediction rules with at least 90% confidence, generated by C5.0 with LOOCV.

Note from Figure 5 that in the majority of cases, learning-based prediction is almost as good as, or better than, profile-based. However, for _205_raytrace and pmd, learning-based prediction is significantly worse. This may be because these benchmarks are unlike the others, e.g. _205_raytrace is significantly smaller than all the other execution traces. Perhaps a greater diversity in the training set would result in more generally applicable squash prediction rules.

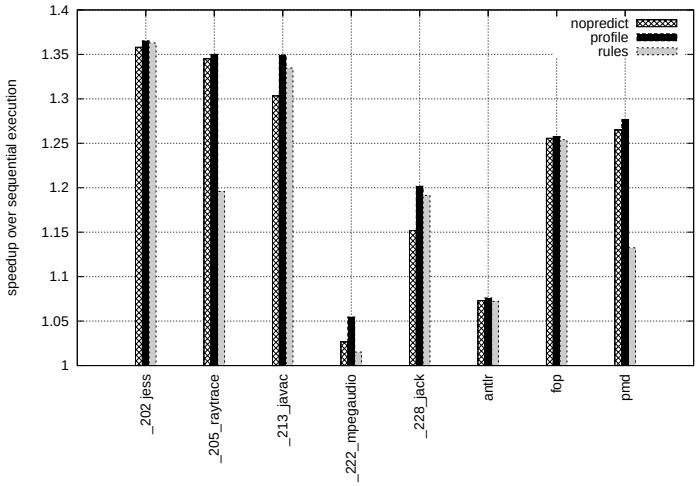

Fig. 5. TLS speedup for various squash prediction schemes

Figure 6 shows the frequencies of TLS spawn and squash events for each benchmark, with each squash prediction policy. For each benchmark, there are three bars: no prediction, profile-based squash prediction and rules-based squash prediction. For each prediction policy, the lower solid bar gives the number of squashes for a benchmark execution, and the upper shaded bar gives the number of thread spawns for a benchmark execution.

The desired impact of squash prediction is to reduce the number of squashes, thus eliminating wasted work. At the same time, the predictor must not have a high *false positive rate*, since this would suppress genuinely data-independent threads and decrease the parallel performance. Figure 6 shows that rules-based

squash prediction reduces the number of squash events in relation to no prediction, in all cases. In the majority of cases, fewest squash events occur with profile-based prediction.

The detrimental effect of a high false positive rate can be seen with the pmd benchmark. The aggressive rules-based squash prediction suppresses many thread spawns, so much so that the number of spawned threads is around 50% of that for the other policies. This reduces the amount of available parallelism in the program execution, this reducing the speedup over the sequential version.

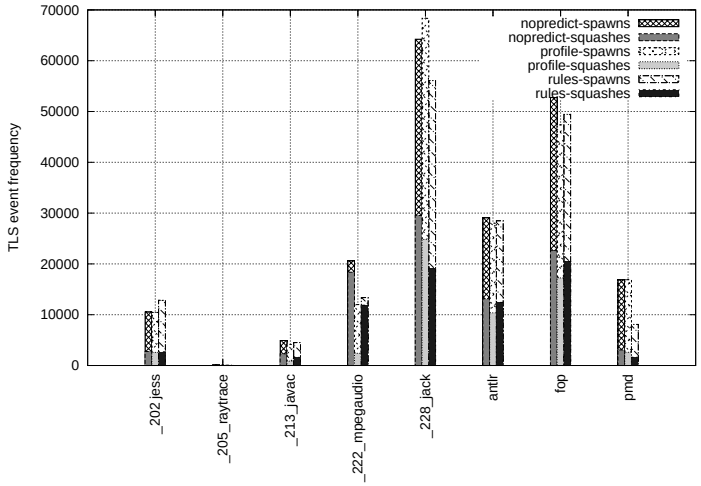

Fig. 6. Frequency of TLS thread spawn and squash events for each prediction policy, (solid bars are squashes, shaded bars are spawns)

5 Related Work

Whaley [4] 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 [5] 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 [22, 10]. In contrast, Dou and Cintra [23] 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 [24] present a hardware lookup table approach that learns which data dependencies 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 is currently intended as a static squash prediction technique, although we are investigating using learning at runtime for dynamic squash prediction.

Warg and Stenstrom [25] 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 use a small TLS event overhead of 10 cycles, which means the overhead is generally negligible in relation to the speculative runlength. (Other researchers adopt 10 cycles as a realistic cost - cites.) In fact, Warg and Stenstrom's problem is largely orthogonal to the squash prediction problem. We could apply similar learning techniques in both cases.

6 Conclusions

This paper has investigated the use of static Java program features for TLS squash prediction, using leave-one-out cross-validation to generate results for previously unseen programs. On the whole, the performance generated from rules-based squash prediction is not overly impressive. In several cases for our benchmarks, it would be better to spawn everywhere, rather than use the squash predictions. One possible inference is that our current set of static features may not *characterize the problem sufficiently* to enable accurate predictions. Alternatively, perhaps we have not selected enough programs to provide good *coverage in the training set* for LOOCV.

There are numerous parameters to tune in our TLS simulator. For instance, we have fixed overhead costs for squashes and thread sizes for consideration as spawn candidates. Additionally, there are parameters in our machine learning setup. We can vary the confidence threshold for selection of rules. We can select rules for both commit and squash events, and vote between them in the event of a disagreement. The irony is that our initial justification for machine learning was

to avoid manually generated heuristics. However now we are intending *to tune parameters* for our customized learning algorithm, instead of for the underlying TLS squash prediction problem.

In future work, we should consider loop-level speculation too. We may be able to use features from existing machine learning studies on loop optimizations, e.g. [26]. We may also progress to consider dynamic features, such as read- and write-set sizes for speculative threads. This may tie in with the theme of *online learning*, integrated into the runtime system with low-overhead profiling and learning costs.

References

1. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium. (2002) 20–29
2. Cintra, M., Llanos, D.: Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In: Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming. (2003) 13–24
3. Quiñones, C., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.: Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In: Proceedings in Conference on Programming Language Design and Implementation. (2005) 269–279
4. Whaley, J., Kozyrakis, C.: Heuristics for Profile-Driven Method-Level Speculative Parallelization. In: Proceedings of the 34th International Conference on Parallel Processing. (2005) 147–156
5. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming. (2006) 158–167
6. Johnson, T., Eigenmann, R., Vijaykumar, T.: Speculative Thread Decomposition Through Empirical Optimization. In: Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming. (2007) 205–214
7. Luo, Y., Packirisamy, V., Hsu, W.C., Zhai, A., Mungre, N., Tarkas, A.: Dynamic Performance Tuning for Speculative Threads. In: Proceedings of the 36th Annual International Symposium on Computer Architecture. (2009) 462–473
8. Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., Sager, D., fook Ngai, T., Fang, J.: Dynamic Parallelization of Single-Threaded Binary Programs using Speculative Slicing. In: Proceedings of the 23rd International Conference on Supercomputing. (2009) 158–168
9. Hu, S., Bhargava, R., John, L.K.: The Role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism. Journal of Instruction-Level Parallelism **5** (2003) 1–21
10. Chen, M., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs. In: Proceedings of the 30th Annual International Symposium on Computer Architecture. (2003)
11. Warg, F., Stenström, P.: Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms. In: Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques. (2001) 221–230

12. Oplinger, J., Heine, D., Lam, M.: In Search of Speculative Thread-Level Parallelism. In: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques. (1999) 303–313
13. : Spec jvm98 http://www.spec.org/jvm98.
14. Blackburn, S., Garner, R., Hoffmann, C., Khang, A., McKinley, K., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, E., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., VonDincklage, D., Wiedermann, B.: The DaCapo Bbenchmarks: Java Benchmarking Development and Analysis. In: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. (2006) 169–190
15. Alpern, B., Attanasio, C., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., Hind, M., Hummel, S., Lieber, D., Litviniv, V., Mergen, M., Ngo, T., Russel, J., Sarkar, V., Serrano, M., Shepherd, J., Smith, A., Sreedhar, V., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM Systems Journal **39**(1) (2000) 211–238
16. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. IEEE Computer **35**(2) (2002) 50–58
17. Xekalakis, P., Ioannou, N., Cintra, M.: Combining thread level speculation helper threads and runahead execution. In: Proceedings of the 23rd International Conference on Supercomputing. (2009) 410–420
18. Singer, J., Pocock, A., Brown, G., Luján, M., Yiapanis, P.: Fundamental Nano-Patterns to Characterize and Classify Java Methods. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications. (2009) 204–218
19. Høst, E., Østvold, B.: The Programmer's Lexicon, Volume I: The Verbs. In: Proceedings of the 7th International Working Conference on Source Code Analysis and Manipulation. (2007) 193–202
20. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: MILEPOST GCC: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit. (2008)
21. Quinlan, R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc. (1993)
22. Wu, P., Kejariwal, A., Cascaval, C.: Compiler-Driven Dependence Profiling to Guide Program Parallelization. In: Proceedings of Workshop on Languages and Compilers for Parallel Computing. (2008) 232–248
23. Dou, J., Cintra, M.: A Compiler Cost Model for Speculative Parallelization. ACM Transactions on Architecture and Code Optimization **4**(2) (2007) 12
24. Cintra, M., Torrellas, J.: Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In: Proceedings of the 8th International Symposium on High Performance Computer Architecture. (2002) 43
25. Warg, F., Stenström, P.: Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. (2003) 12.2
26. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. (2009) 177–187