Speculative Optimization Using Hardware-Monitored Guarded Regions for Java Virtual Machines

Lixin Su and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
lsu@cae.wisc.edu, mikko@ece.wisc.edu

Abstract

Aggressive dynamic optimization in high-performance Java Virtual Machines can be hampered by language features like Java's exception model, which requires precise detection and handling of program-generated exceptions. Furthermore, the compile-time overhead of guaranteeing correctness of code transformations precludes many effective optimizations from consideration. This paper describes a novel approach for circumventing the optimization-crippling effects of exception semantics and streamlining the implementation of aggressive optimizations at run time. Under a hardware-software hybrid model, the runtime system delineates guarded regions of code and specifies a contract—in the simplest case, one that requires exception-free execution—that must be adhered to in order to ensure that the aggressively optimized code within that region will behave as the programmer expects. The contracted runtime condition is assumed to be true, and code within a guarded region is aggressively optimized based on this assumption. Hardware monitors for exceptions throughout the region execution, and undoes the effects of the guarded region if an exception occurs, re-executing the region with a conventionally optimized version. Since exceptions are very rare, code can be optimized as if optimization-crippling conditions did not exist, leading to compile time reduction, code quality improvement, and potential performance improvement up to 67.7% and averaging 15.9% in our limit study of a set of Java benchmarks.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – Compilers, optimization, Run-time environments, Code generation
General Terms *Language, Performance, Algorithms*
Keywords Virtual machines, Transactional memory, Speculative processors, Java, precise exceptions, speculation

1. Introduction

Dynamic optimization techniques play an increasingly important role in delivering high performance for many classes of applications. Advanced runtime systems that incorporate dynamic optimizers have numerous fundamental advantages over the classical static approach to compilation, since knowledge available only at runtime can be exploited to focus optimization activity on important portions of the program, avoiding code analysis and optimization time overhead for other unimportant or less important portions of the program. In fact, many previously-unattractive yet powerful optimizations that require expensive global analyses for correctness have become tractable due to the scope-limiting effect of runtime profile information. Furthermore, advanced execution profiling techniques can supply detailed control flow and even value profiles that can be used to specialize the generated code for the common case, often leading to code quality that is beyond the reach of conventional static optimizers--even if static optimizations are applied indiscriminately, without regard for the computational cost of program analysis and optimization. Numerous high-quality dynamic optimization environments have been developed and even released for the research community's use (e.g. [3][5][10][17][25]).

However, aggressive dynamic optimization in high-performance Java Virtual Machines (JVMs) is often inhibited by an otherwise powerful and useful language feature: Java's exception model. The Java standard specifies a powerful and programmer-friendly model for specifying, raising, and explicitly handling program-induced exceptions. Unfortunately, the potential for exceptions dramatically complicates and often cripples common algorithms for aggressive code optimization. As a result, Java suffers performance losses, even though exceptions (by design) occur rarely--if ever--in the steady-state execution of most programs. Due to Java language semantics, potentially excepting instructions (PEIs) induce optimization barriers, since they must remain in their original program order, and also impede code motion, register assignment, and other desirable optimizations due to their language semantics.

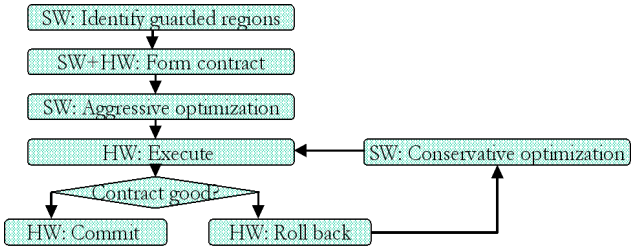

Figure 1 Hybrid framework for aggressive dynamic optimization

This paper describes a novel speculative semantic for aggressive dynamic optimization for programming languages like Java, and demonstrates how such an approach can be used to overcome both the compile- and run-time overhead brought about by Java's exception model. Figure 1 illustrates our approach: the software identifies and delineates a guarded region (e.g. a function or loop body) to the hardware, specifies the conditions that must be met for the guarded region to satisfactorily commit its execution, and then aggressively optimizes the region's code, taking advantage of the contracted conditions. The hardware monitors the

VEE'07 June 13–15, 2007, San Diego, California, USA.

execution continuously, and allows the region to commit if the conditions are met, or, in case of failure, rolls back and instead executes conservatively optimized code. In the general case, runtime conditions could be selected from a broad set of possibilities: control-flow outcomes, detection or absence of memory dependences, absence of garbage collection events, and so on. However, in this initial paper, we consider only a very simple condition—exception freedom—and show that it has the potential to substantially simplify optimization algorithms, reduce compilation overhead, and speed up runtime execution.

A primary benefit of our approach lies in its ability to enable *speculative optimizations*. If the runtime conditions are specified appropriately, code transformations and optimizations need not be fully and provably correct, as long as the runtime conditions catch any corner cases that the optimizer failed to accommodate. This provides two desirable effects: first of all, more aggressive optimization is possible, since transformations that the compiler cannot prove are always safe can still be productively employed. Second, the optimizations themselves can be implemented more efficiently. This latter benefit is extremely important in dynamic optimization, where the compile-time overhead of many traditional optimizations precludes their use, since it could easily overwhelm the run-time benefit. In fact, this benefit opens up a new realm of opportunities for investigating compile-time-efficient, *mostly-correct* code transformations that guarantee correctness through specifying modest run-time invariant conditions that hardware can easily guarantee. We describe two such algorithms in Section 3.

The scope of each speculative optimization is bounded by its enclosing guarded region. In order to enable high-quality optimization, the regions should be reasonably large, hence enabling greater freedom for code motion and other global optimizations. However, the regions should not grow uncontrollably; since hardware has limited capability for isolating the effects (e.g. buffering memory writes) of a region until the execution successfully reaches the region end. We discuss the trade-offs involved in region selection and propose several simple but effective heuristics.

While the proposed approach can be applied to a broad class of speculative optimizations, in this initial work we focus on alleviating the optimization-crippling effects of just two potentially exception-causing instructions: array bounds checks and null pointer checks. We have found that even with a state-of-the-art check elimination mechanism included in our baseline JVM [6] these checks cause substantial performance overhead for our benchmarks, up to 67%. Part of the performance loss is caused by the overhead of conducting the checks themselves, but we have also found that the presence of the check instructions (PEIs) in the optimizer's intermediate representation (IR) substantially impedes other, seemingly unrelated optimizations and thus affect the overall effectiveness of the whole optimization flow. By aggressively eliminating both types of checks within our speculative optimization environment, we are able to reap the vast majority of the potential benefit of eliminating these checks.

In the remainder of the paper, Section 2 describes how Java's precise exceptions affect runtime performance; Section 3 introduce new lightweight speculative optimization algorithms; Section 4 discusses a few guarded region placement heuristics, as well as the hardware support; Section 5 describes our experimental methodology; Section 6 presents results; Section 7 conducts a literature survey; and Section 8 concludes the paper.

2. How Exceptions Impede JVM Performance

The Java programming language defines four types of exceptions - errors, asynchronous exceptions, runtime exceptions, and checked exceptions [19]. The latter two exceptions need to follow the precise exception model. This means that exceptions must be thrown in the same order as specified by the original (unoptimized) program and that the program state observable at the entry of an exception handler must be the same between the original and the optimized code. Potentially-excepting instructions (PEIs) that can throw runtime and checked exceptions are nearly ubiquitous in Java programs, although the exceptions are rarely thrown. The PEIs, requiring precise handling, can severely limit the compiler's ability to perform many advanced optimizations such as instruction scheduling and dead code elimination.

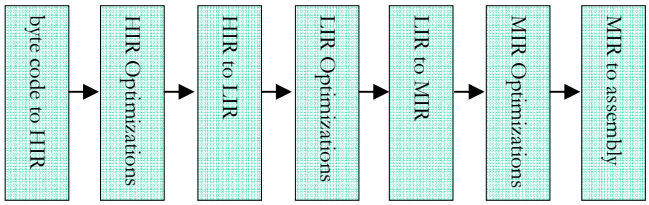

Figure 2 Jikes optimizing compiler's optimization flow.

The most common PEIs in real applications are check instructions, which include null checks (against null pointer instantiation), bounds checks (against array out of bounds memory accesses), zero checks (against division by zero), store checks (against an incompatible object reference saved in a reference array), and checkcast (against incompatible type cast). Among these check instructions, null checks and bounds checks account for a majority of them. Null and bounds checks are usually treated as single instructions in many optimization phases in order to enable more optimization opportunities. At the beginning of the optimization flow, certain optimizations such as local common subexpression elimination (CSE) and local bounds check elimination (BCE) are used to remove some checks.

However, many checks are still present in the subsequent optimization phases and limit the potential for optimization. In the middle or later stages of the optimization flow, a check instruction is expanded to a branch instruction and a call instruction. The call instruction is executed to throw an exception if the check fails. In the end of the optimization flow, a majority of explicit null checks can be eliminated by combining them with loads/stores, whenever the virtual memory system can be configured to raise exceptions for accesses to page 0. These exceptions can be caught by the JVM and handled as null check violations.

The optimization flow for Jikes [3], shown in Figure 2, is to illustrate the check instruction handling process in a JVM. There are three levels of IR - high-level IR (HIR), low-level IR (LIR) and machine-level IR (MIR). All three levels have multiple optimization phases. At HIR generation, check instructions are separately generated from their associated instructions. Their ordering is strictly maintained and creates performance constraints for later optimizations. Some checks can be statically proven redundant and removed while many simply propagate

through the following optimization phases, which increases the IR size and the compile time. In one MIR optimization phase (NullCheckCombining), null checks are combined with loads/stores in a basic block if there are no PEIs between them.

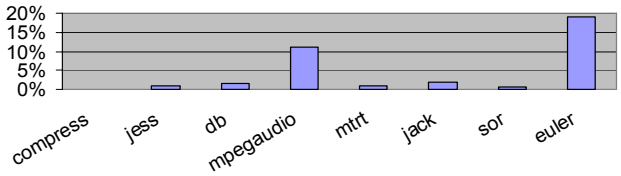

Figure 3 Performance improvement due to NC elimination at the beginning of the optimization flow.

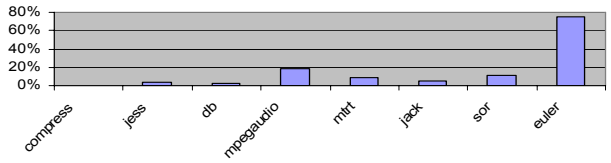

Figure 4 Performance improvement due to BC elimination at the beginning of the optimization flow.

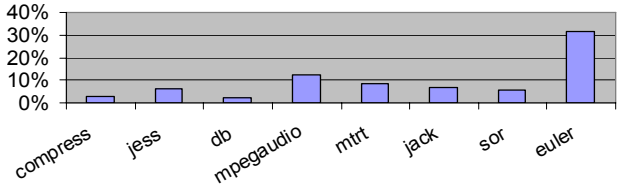

Figure 5 Compile time increase due to checks.

To motivate our earlier claim that the presence of PEIs impedes other, seemingly unrelated optimizations and affects the overall effectiveness of the optimizing compiler, we modified Jikes to eliminate null checks the beginning of the optimization flow instead of at the later NullCheckCombining stage (refer to Section 5.2 for experimental details). The performance improvement of early null check elimination is shown in Figure 3. The unmodified Jikes can combine about 88% of null checks with loads/stores at the NullCheckCombining stage in our benchmarks. In the baseline we went further to delete the remaining 12% after the NullCheckCombining stage (This deletion does not lead to any performance improvement in our benchmarks). However, the baseline still suffers performance losses compared with null check elimination at the beginning of the optimization flow. The losses are quite significant in some benchmarks. We tried to vary the optimization phases in the optimizing flow, e.g. by disabling optimizations in SSA (disabling SSA entirely caused slowdowns of 5%-15% overall) and found that the performance loss induced by null checks almost disappeared for mpegaudio, was slightly lowered for euler and sor, and was greatly lowered for other benchmarks. We conclude that certain optimization opportunities and the overall effectiveness of the optimizing compiler are hindered by the presence of null checks.

The ideal case performance improvement due to bound check elimination at the beginning of the optimization flow is shown in Figure 4. Bound checks impede performance even more than null checks. We also tried to vary the mix of optimization phases in the optimization flow by the deletion of optimization phases in SSA and found that the performance losses due to bounds checks for our benchmarks fluctuated compared with the case where SSA optimizations existed. The performance impact for mpegaudio, mtrt, and euler lowered by about 30 per cent while remained within 10 per cent for other benchmarks.

Figure 5 shows the compile time increase due to null checks and bounds checks; the IR size increase induced by the checks can substantially increase compilation overhead and slow down execution. In the baseline we simply delete both checks at the beginning of the optimization flow and thus there is no compilation overhead from these two checks.

3. Speculative Optimization Opportunities

As described in Section 1, a hybrid hardware/software optimization framework supporting speculative optimization within guarded regions can overcome many exception-related impediments to JVM optimization. With hardware support for undoing the effects of guarded regions that fail condition checks, speculative optimizations can be aggressively performed even under Java's strict exception model. Furthermore, hardware-assisted recovery enables use of efficient algorithms with relaxed correctness requirements, reducing compilation overhead.

In this initial paper, we assume the contracted runtime condition as exception freedom and tackle null checks and bounds checks as examples to illustrate opportunities for our approach. This section describes the algorithms we have implemented in Jikes to reduce the overhead of these checks.

```
Compute dominator/post-dominator information;
foreach basic block (BB) {
  boolean isInRegion = false;
  if ((a region start is in this BB's dominators &&
       there is no region end in between) &&
      (a region end is in this BB's post-dominators &&
       there is no region start in between))   isInRegion = true;
  if (isInRegion) eliminate all null checks in this BB;   }
```

Figure 6 General algorithm for null check elimination.

3.1 Null checks (NCs)

Most NCs can be safely eliminated if they are known to be within the scope of a guarded region, since the JVM can rely on the virtual memory subsystem to detect null references, and can replay an unoptimized version of the guarded region to regenerate the exception precisely. NCs should be removed as early as possible in the optimization flow, in order to minimize their detrimental effects on the optimization process. However, they cannot be eliminated until the guarded regions have been identified. Once guarded regions are identified, region start/ends are inserted in the intermediate representation (IR) to mark the boundaries of the identified regions. Section 4.2 discusses several guarded region placement heuristics. In all heuristics considered in this paper, regions are identified right after the first IR generation stage and we can determine very early in the optimization flow whether basic blocks in a certain method are enclosed within a guarded region. In general, dominator and post-dominator information are needed to determine if a NC is within a guarded region (dominators and post-dominators are already computed for other optimization purposes, hence incurring no additional overhead). First, we examine a basic block's dominators to check that there is a *region start* and that there is no *region end* between the region start and the basic block. Second, we examine a basic block's post-dominators to check that there is a region end and

there is no region start between the basic block and the region end. The basic block is within a guarded region if both conditions are satisfied; therefore, its NCs can be speculatively removed. The algorithm is shown in Figure 6. Since the early removal of NCs relies on the execution of the correspondent loads/stores, cautions need to be taken to prevent data-flow-dead code elimination algorithms from optimizing away dangling loads/stores. An alternative solution is to replace loads that can be optimized away back with null checks. It is worth noting that the computing, on which data-flow-dead loads/stores are data-dependent, cannot be optimized away with or without the speculative NC elimination algorithm.

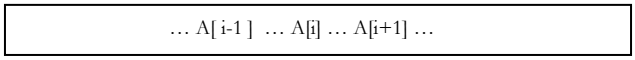

Figure 7 Example for local BC elimination.

3.2 Bounds checks (BCs)

BCs can be removed when they are proved to be subsumed by another BC. Our baseline JVM incorporates ABCD[6], a state-of-the-art BC elimination algorithm. However, there are many remaining BCs that incur performance overhead and impede aggressive optimization. We develop a speculative local BC elimination algorithm and a speculative loop-based global BC elimination algorithm based on the loop monotonic statement detection algorithm proposed by S&G [28]. We describe more aggressive loop-based BC elimination algorithms based on application characteristics. We describe our algorithms in the context of upper bounds and the algorithms' duals (complementary versions) can easily handle lower bounds.

1. Convert an array bounds check (BC) A[index] to a tuple.
   - If index is a register (r_1) defined using move, trace back to the defining statement (r_n=…) that is not a move instruction and start from 1 to create a tuple for A[r_n].
   - If index is constant, convert to <A, special_reg, constraint>.
   - If index is defined by an addition/subtraction that involves a register (r_1) and a constant, create a tuple <A, r_1, constraint>.
   - If index register is defined by a phi instruction or it is a parameter register, create a tuple <A, index, constraint>.
2. Check the tuple in its specific group.
   - If the group does not exist, create its group and update the group's current BC.
   - If the group exists and its current constraint is larger than or equal to this tuple's, mark the BC redundant.
   - If the group exists and its current constraint is smaller than this tuple's, mark the previous BC redundant and update this group's current BC to the new one.

Figure 8 SSA-based speculative local BC elimination algorithm.

3.2.1 SSA-based local bounds check elimination

This clean, general, and lightweight algorithm is designed to speculatively eliminate redundant bounds checks within a basic block (BB). An example is shown in Figure 7. The three array accesses are distributed in a basic block. A non-speculative local BC elimination algorithm cannot safely remove the bounds checks for A[i-1] and A[i]. The pure software-based speculation such as check promotion cannot efficiently handle this either. Such speculation needs to promote the strictest check, in this case the check for A[i+1], above this BB, which might not be feasible. Some software speculation requires a replication of the BB with one version containing all checks while the other dropping the checks. This leads to code bloat and can complicate JVM performance tuning. Some might propose to use a stub function that activates the JVM to regenerate this BB with checks when the promoted check fails at runtime execution. One stub function per BB and the BB regeneration information needed for this stub function can easily introduce enough overhead to offset the gain from speculatively removing some bounds checks.

With our approach, the two bounds checks can be speculatively removed without introducing any runtime overhead, since we no longer have to maintain their relative order. Our algorithm reduces the code size and adds zero runtime overhead in the commonly executed code.

The algorithm's prerequisites are SSA and def/use chains. The algorithm is more efficient if local common subexpression elimination (CSE) is performed in advance. A tuple <array ref register, index register, constraint> is used to represent a bounds check. For example, A[i-1] is converted to <A, i, -1>, A[i] to <A, i, 0>, and A[i+1] to a tuple <A, i, 1>. Different bounds checks are compared against each other regardless their program order. Tuples belong to the same group if their array ref registers and index registers are the same. In the same group only the bounds check with the largest constraint is not redundant. Bounds checks with constant array indexes are converted to tuples belonging to a group with the array ref register and a special index register. The algorithm is shown in Figure 8. In tracing back the index register's def chain we only consider moves and additions/subtractions involving a register and a constant; other operations with more general forms can be included later.

3.2.2 Loop-based global bounds check elimination

Our loop-based algorithm is superior to loop versioning [24][17], a software-based speculative technique to remove BCs in loops, since loop versioning adds runtime execution overhead, increases the code size, and only works for specific loops. It can limit the effectiveness of other optimizations such as loop unrolling and dramatically increase the difficulty of the JVM performance tuning. Our algorithm adds zero runtime overhead, does not increase code size, and is applicable to all loops.

The algorithm is developed based on S&G's loop monotonic statement detection algorithm. A loop monotonic statement is one that always increases/decreases a variable during the loop iteration. S&G characterizes statements in a loop as monotonic, invariant and chaotic. Loop monotonic statements can be divided into three categories: basic, dependent and cyclically monotonic statements. Their definitions can be found in [28].

Our algorithm focuses on the monotonicity of variables instead. The overview of the algorithm is given in Figure 9. The algorithm requires loops, dominators, and def chains to be computed first. Among loops, inner ones are processed before outer ones.

In step 1 variables used as array subscripts are identified. The monotonicity analysis targets such variables instead of every variable or statement in a loop in order to reduce the analysis cost.

Step 2 finds all the variables necessary for the monotonicity analysis of array subscript variables. The traversal of the def chain for an array subscript variable stops when the variables in the def chain have no in-loop definitions. Additional early traversal termination conditions can be introduced to reduce

computation cost. First, a variable in the def chain has more than two in-loop defining statements with different operators. Second, one of the defining statements is a load instruction or a call instruction.

1. Find variables used as array subscripts.
2. Traverse the def chain of array subscript variables until variables with no in-loop definitions.
3. Construct data dependence graph (DDG) for the identified variables.
4. Characterize variables into potentially basic monotonic variables, potentially dependent monotonic variables, and potentially cyclically monotonic variables.
5. Prune potentially cyclically monotonic variables and mark them chaotic.
6. Identify the initial values of variables positive, negative, either, non-negative, or non-positive constants.
7. Derive monotonicity of potentially basic monotonic variables.
8. Derive monotonicity of potentially dependent monotonic variables in the order that each variable being processed is only data dependent on variables that have already been processed.
9. Move array bounds checks outside the loop if the subscript variables are invariant or monotonic and the array reference pointer can be moved outside the loop.

Figure 9 Loop-based speculative bounds check elimination.

Step 3 is to construct a data dependence graph (DDG) for all the variables to be analyzed.

Step 4 marks the tree root nodes in the DDG as potentially basic monotonic. It then identifies the Strongly Connected Components (SCCs) in the DDG and other variables that are data dependent on at least one variable in a SCC. These are potentially cyclically monotonic variables. In step 5 we mark such variables chaotic to avoid analyzing them in later stages.

Step 6 identifies the initial values of the variables if they have initial values upon the entrance of the loop. This is not a trivial task as it relies on dominators and post-dominators to sort out the relationships of different definitions of a variable outside the loop. We notice that three special cases can cover many cases in programs. Case 1 is that the variable is assigned to a constant in the immediate dominator of the loop. In Figure 10, variable j falls into this case for the inner loop. In case 2, a variable only has one definition outside the loop and it is in a dominating basic block other than the immediate dominator of this loop. Variable k for the inner loop in Figure 10 is an example. In case 3, the variable is initialized as a constant upon the entrance to the outer loop and gets increased/decreased by a constant in the outer loop iteration. Variable i for the inner loop is an example for the third case. The three special cases can significantly reduce the computation cost while capturing most opportunity.

```
for (int i = 1, k = 1; i < n; i++)
   for (int j = 1; j < n; j++)     { A[ i ]; A[ j ]; A[ k++ ]; }
```

Figure 10 Example for variable initial value identification.

Step 7 and 8 derive the monotonicity of potentially basic monotonic and potentially dependent monotonic variables. Here, our definition of monotonicity also includes invariance, which is different from S&G. We use their algorithms to characterize variable's monotonicity. We also add support for instructions such as move, neg, and shift.

Step 9 moves BCs outside the loop if possible. We avoid replicating the first iteration and the last iteration to prevent code bloat. The final BC is checked after the loop if the subscript is monotonically increasing; the initial BC is checked before the loop if the subscript is monotonically decreasing. Both BCs need to be checked if the subscript is monotonic. The BC can be moved to either place if its subscript is a loop invariant. Necessary compensation may be applied for the subscript of a BC that is moved out of the loop. We also rely on speculation to simplify BC motion as illustrated in Figure 11. Two pad basic blocks (BB2p and BB4p) and a replicated branch in BB2p need to be generated to guarantee the correctness of moving BCs outside the loop as shown in part (b) in if no speculation is used. BCs can not be directly moved to BB2 and BB4 as they might not be executed in the original loop. However, this worry is unnecessary since almost all busy loops execute at least one iteration. Moving BCs to BB2 or BB4 will rarely cause misspeculation. With support for guarded regions we can safely put BCs in BB2 and BB4 assuming that replay rarely happens. Therefore, we can keep the original loop structure. BCs can still be moved out of the loop even if they are executed conditionally. However, this is more likely to cause guarded regions to roll back.

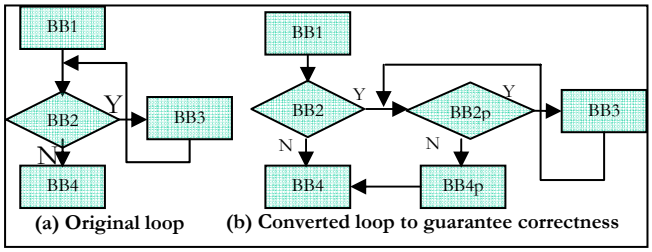

Figure 11 Simplified loop manipulation with guarded regions.

3.2.3 More loop-based global bounds check elimination

A typical array access pattern we have seen in real applications that cannot be captured by the algorithm in Section 3.2.2 is shown in Figure 12. The programmer sets an upper bound for the value of an array-indexing variable. In the example the programmer assumes that j can not be larger than 15. In this case the speculative optimizing compiler can safely assume that array A has a size most likely larger than 15. Therefore, a BC A[15] can be placed before the loop and the BC in the loop can be eliminated. This is an example of slightly riskier speculation. The dynamic compiler can not guarantee that array A has a size larger than 15 but it makes an educated guess that this should be most likely true. Therefore, it decides to speculatively hoist the BC.

Many applications access multidimensional arrays, as shown by the example in Figure 13. For such arrays the loop-based algorithm mentioned in Section 3.2.2 has limited effectiveness. The loop-based algorithm can move the BC for A[i] outside the outer loop. The BC involving variable j can only be moved outside the inner loop. For applications with many multidimensional array accesses further unexploited opportunity remains. One possible solution is to provide hardware support for register min/max value monitoring, and replay a guarded region if a particular register reaches a value that exceeds the array bounds. This approach works well for the example in Figure 13 and for most other array access patterns. In the example, two register values need to be watched - the min value v1 for the array length of A[i] and the max value v2 for variable j. The BC involving j can be

completely eliminated in the loop. Then v1 is compared to v2 after the loop. An exception is thrown and the guarded region replays if v1 is less than v2. Register value monitoring can even be applied to array BCs involving non-monotonic variables. This solution requires the processor to have enough registers to hold each monitored variable. The current IA32 processor only has 8 integer registers and register spills can occur, complicating code generation and potentially causing performance hazards. However, 64-bit AMD64/EMT64 extensions to IA32 have 16 registers, Itanium has 128 registers, and Power5 has 32 registers. Hence, future processors will have more registers and register spills will become less of an issue for register value monitoring.

```
for (int i = 0; i < n; i++) {.. j = j + 1 & 15; … A[j]; … }
```

Figure 12 Code not captured by the Section 3.2.2 algorithm.

```
for(int i = 0; i < n; i++)   for(int j = 0; j < m; j++)  {…A[ i ][ j ]… }
```

Figure 13 Example code with continuous 2D array accesses.

```
int [ ][ ] A = new int [2][ ]; A[0] = new int [3]; A[1] = new int [4];
```

Figure 14 An asymmetrical array.

A further complication arises in the presence of asymmetrical arrays. An example is shown in Figure 14. In the example A[0] and A[1] have different array lengths. A possible solution could be given from either the language level or the implementation level. The language can specify symmetrical arrays. The JVM can also include a flag in its array implementation to indicate an array is symmetrical. In addition, asymmetrical multidimensional-arrays rarely occur in real applications, which should ease the possible implementation of the proposed solution.

3.2.4 Discussion

After our BC elimination algorithms are applied, array accesses are still protected by a BC. The execution of the BC leads to the load of the array length (typically at memory address array_pointer+4), which would access virtual memory page 0 if the array point was null. Therefore, NC elimination of array pointers can safely co-exist with BC elimination.

The loop-based BC elimination algorithms do not require loop termination analysis for loops that generate writes to be buffered. Such loops account for almost 100% of the loops in real applications. If a loop is infinite, the write buffer will eventually overflow and lead to a region replay. For unterminated loops without any writes, there is no problem, since the hardware has the capacity to monitor the loop's execution essentially forever.

Possible integer overflow for array subscript variables is not a problem either. We simply require the hardware to detect integer overflow events (the detection hardware is typically present in the ALU anyway to support extended precision arithmetic operations) and flag a potential exception. If the program is within a guarded region when the potential exception occurs, the hardware rolls back the execution and re-executes conservatively-optimized code that does not rely on absence of integer overflow. We have not observed any cases where this actually occurs in our benchmarks.

4. Guarded Region Placement

In our fully automated dynamic optimization environment, guarded regions must be identified and delineated efficiently. This section describes simple heuristics for placing guarded regions. In our approach the program analysis cost is minimized and hardware provides the capability of offsetting the potential performance degradation incurred by the simplistic analysis. In this section we first describe the hardware support for our cooperative automatic guarded region placement and then show a few low overhead algorithms suitable for a dynamic optimizer.

4.1 Hardware support

Hardware that supports our speculative semantic simply needs to support buffering of state updates and conditional rollback or commit. We discuss some possible approaches in Section 7.2, but leave detailed evaluation to future work. The hardware also needs some Instruction Set Architecture (ISA) extensions, including at minimum a region start (reg_start) and region end (reg_end) instruction. However, this limited ISA support may not be sufficient for effective region placement. On one hand, compilers need to create large regions to increase optimization scope. On the other hand, hardware can only efficiently buffer a limited amount of speculative execution (arguably a few kilobytes to a few tens of kilobytes) if buffering is the choice of the hardware implementation. Large guarded regions have more writes to buffer and are more likely to cause buffer overflow. Buffer overflow forces a region replay and has a significant overhead. Therefore, this dilemma seemingly forces the compiler to perform compute intensive analysis to create regions of ideal size.

We advocate additional ISA support -- conditional *region start/end*, which are still speculative optimizations barriers but serve as hints to the hardware with respect to the commit of the on-flight region and the start of a new region. If the hardware buffering capability is near a high-water mark, hardware should end (commit the writes of) the region and start a new one. If there is still plenty of room to buffer more activity, hardware should treat it as a NOP, basically collapsing together the region currently in flight with the next one. This conditional region start/end allows compilers to insert region boundaries somewhat indiscriminately, using the easiest or least computation-intensive placement algorithm. Hence, the computationally difficult problem of automatic region placement no longer has to be solved precisely, since hardware can optionally discard unnecessarily frequent boundaries. Conditional region start/end relies on hardware to commit without stalling the execution engine, which should not be a problem in HTM proposals and other speculative hardware proposals as they allow commits and execution concurrently. Very rarely, the execution engine could stall due to commits and this could be performance hazardous.

4.2 Static region placement heuristics

In this section we describe three static region placement heurisitics with nearly zero compile-time overhead. These heuristics are early research investigations and more sophisticated placement heuristics that can utilize some static analysis such as inlining analysis will be our future exploration.

4.2.1 Leaf function based placement (Leaf)

This heuristic simply treats leaf functions as regions. It assumes that an application spends most time in leaf functions. A leaf function is an application function that does not contain any function call to another application function after inlining has been executed. A leaf function can have Java library function calls since a high performance JVM usually has its own proprietary library implementation and can limit external effects, e.g. the number of writes, of a library function call or at least calls

to a majority of library functions. A region start is placed at the entrance of a leaf function and a region end is placed at the exit as shown in Figure 15(a). This approach has two drawbacks. First, there is a fair amount of execution time in non-leaf functions. Second, some leaf functions can generate more write traffic than the hardware's limited buffering capability. This can lead to unnecessary replays due to buffer overflow. However, this heuristic generates the smallest number of regions among the three heuristics and has the additional advantage of ruling out the possibility of nested regions.

```
leafFunction {
  region_start;
  …
  region_commit;  }
```
(a)

```
callerFunction {
  region_start;
  calleeFunction();
  region_commit;   }
calleeFunction {
  region_conditional_commit;
  region_conditional_start;
  …
  region_conditional_commit;
  region_conditional_start;    }
```
(b)

```
callerFunction {
  region_start;
  calleeFunction();
  region_commit;   }
calleeFunction {
  region_conditional_commit;
  region_conditional_start;
  …
  loop1 {
    region_conditional_commit;
    region_conditional_start;
    loop2 { … }
    region_conditional_commit;
    region_conditional_start; }
  …
  region_conditional_commit;
  region_conditional_start;    }
```
(c)

Figure 15 Examples for static placement algorithms.

4.2.2 Caller & callee based placement (C&C)

This approach, illustrated in Figure 15(b), tries to extract speculative optimization opportunities in all functions. Upon the entrance and the exit of a function it ends the current region and starts a new one. Thus all the code in every function is enclosed within a guarded region. The conditional region end/starts are used here to reduce unnecessary commits and starts. When such instructions are executed, hardware commit occurs only if buffer space is nearly full. Otherwise these instructions are treated as NOPs in hardware and the current and next regions are spliced together. One drawback is that C&C can force retention of a BC in loops with a function call, since the call forces a region boundary and no BC can be moved across this boundary.

4.2.3 Caller/callee/innermost-loop placement (CCIL)

This scheme is more aggressive than C&C and it can further break down multilevel loops with lots of write traffic, which occur fairly frequently in scientific benchmarks. The current region is conditionally ended and a new one is conditionally started both before and after an innermost loop. CCIL also relies on conditional region end/starts to reduce unnecessary commits. CCIL creates more small regions than C&C since there can be many multilevel loops in applications that do not generate many writes. An example of CCIL is shown in Figure 15(c). CCIL prevents BCs in outer loops from being moved out of outer loops. However, the innermost loops are the hottest and moving BCs out of such loops can still lead to a significant performance gain.

4.3 JVM support for region placement

For the leaf placement heuristic a JVM only needs to utilize the hardware's replay mechanism to roll back to the architected state prior to the execution of the excepting method. For C&C and CCIL a JVM needs a full-blown on-stack replacement (OSR) [11] to switch from the optimized to the upoptimized code in the middle of a method. OSR is a known technique implemented in many JVMs. In our context, the existing OSR technique simply needs to be expanded to utilize hardware replay ability.

5. Experimental methodology

5.1 The challenge of quantitative evaluation

Quantitative performance evaluation of dynamic optimization techniques for future hardware in execution-driven simulators presents a steep challenge. A JVM interacts closely with the operating system and the memory system. Therefore, a full system simulator is preferred to faithfully show the workload performance. A typical full system simulator runs at 20 KIPS. A typical Java workload may need a warm-up period of five minutes for a JVM to collect runtime information and reoptimize code. Simulating a 1 BIPS processor running for five minutes means 5 min x 1 BIPS x 60 sec/min = 300B instructions. At 20 KIPS simulation speed this would take 174 days, which is not feasible.

5.2 Our evaluation methodology

The main goal of this paper is to illustrate the performance improvement opportunity that can be realized by the application of lightweight speculative optimizations within guarded regions. In our evaluation, no exception is ever thrown, and no replay is ever required. We employ hardware performance counters to provide useful information and insights about the possibility of automatic region placement and its implications for hardware. Therefore, a thorough qualitative evaluation, using instrumented execution on a native machine, provides not only a quick turnaround time, but also reasonably accurate performance estimates that are more than adequate for achieving our goals.

Experiments are performed with Jikes RVM v2.3.4 on a 2.4GHz Pentium4 based uniprocessor machine with 1GB memory and Redhat Linux 2.4.22.

Jikes is built with production configuration. Methods are directly compiled at opt2 by the optimizing (opt) compiler, which shows the impact of speculative optimizations and leads to a quick and easy comparison between the baseline and the optimized version.

Table 1 Benchmark Information

Benchmarks	Description	Run time ms
Compress	LZW compression program	5959
Jess	NASA rule-based expert system	2835
Db	Data management benchmark	15740
mpegaudio	MPEG-3 audio codec	5040
Mtrt	Program ray-tracing an image	2765
Jack	Real parser-generator	416
Sor	Successive over-relaxation algorithm	4190
Euler	Computational fluid dynamics	2600

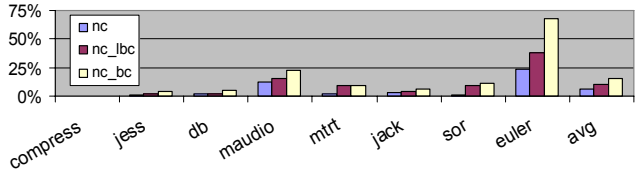

Figure 16 Speedups for perfect region placement. Here, nc = speculative NC elimination; nc_lbc = nc + speculative local BC elimination; nc_bc = nc_lbc + speculative loop-based BC elimination.

We use the SPECjvm98 benchmarks (except *javac*) [32] and two benchmarks in Java Grande [30]. The benchmark information is

shown in Table 1. We follow the run rules and run benchmarks multiple times to report the best numbers.

6. Results

This section presents our evaluation results. It shows that the proposed speculative optimizations can improve performance with perfect guarded region placement. We also evaluate the compile-time overhead of the implemented speculative algorithms. Finally, we show that the proposed automatic placement algorithms can achieve a good percentage of the potential speedup from perfect region placement. We also show that the proposed hardware support, conditional region end/start, can be a key to the success of speculative optimizations and different placement schemes.

6.1 Perfect region placement

By perfect region placement we mean that guarded regions can be ideally placed so that all possible speculative optimizations can occur within a region. An example of perfect region placement is treating the whole application as a guarded region, hence assuming the hardware to have an effectively unbounded buffering capability. In this situation we can apply our speculative algorithms without worrying about region boundaries. In Figure 16, we show the speedups of the benchmark suite due to the application of speculative algorithms incrementally. The average performance increases from 5.7%, 10% to 15.9% with the addition of speculative NC elimination, local BC elimination, and global BC elimination. Compress's performance is not affected by our algorithms. Compress has about 60 BCs in total and its performance critical BCs can not be eliminated by our algorithms. There could be a speedup of more than 12% if such BCs could be speculatively eliminated. If register min/max value monitoring hardware (see Section 4.1) was available, these hot BCs could also be captured.

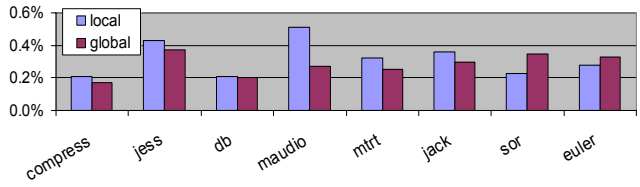

Figure 17 BC elimination algorithms' compilation overhead among the overall compile time.

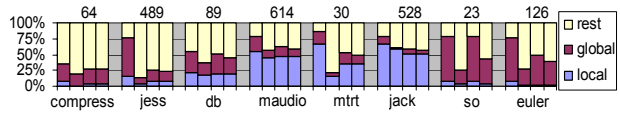

Figure 18 Percentage of BCs removed by local and global algorithms. The numbers at the top are the number of BCs for each benchmark. The four bars are perfect region, leaf, C&C, and CCIL. The three series, local, global, and rest, correspond to BCs removed by the local algorithm, removed by the global algorithm, and untouched.

6.2 Compile-time overhead and coverage

Our speculative algorithms are lightweight. The NC elimination algorithm iterates through basic blocks and removes NCs after a method is identified to be within a guarded region. With perfect region placement and the region placement algorithms, a method is either in regions or not. Therefore, the NC elimination algorithm introduces almost zero overhead.

The BC elimination algorithms are also very efficient. The local one and the global one account for no more than 0.51% and 0.37% of the overall compile time, as in Figure 17.

The percentages of BCs removed by algorithms are shown in Figure 18. The algorithm coverage is high. In perfect region placement, the coverage is more than 70% except for db and compress. In db the hot BCs are captured while in compress they are not captured. The three static region placement algorithms can capture many BCs captured by perfect placement.

6.3 Automatic region placement

The performance improvement, the region size distribution, and the number of regions and the average region size for the three automatic placement algorithms are shown in Figure 19, Figure 20, and Table 2. We measure the region size in terms of the number of writes generated. The number of regions and the average region sizes were collected with performance counter support [31] integrated into Jikes.

The three automatic region placement algorithms can effectively extract the performance improvement achievable by perfect placement, as illustrated in Figure 19. The effectiveness of Leaf depends on the fraction of program execution time in leaf functions. For the benchmarks with most execution time in leaf functions, Leaf extracts almost all opportunity. C&C typically performs better than Leaf since it factors in non-leaf functions. It does not perform as well as perfect region placement because some bounds checks can not be moved outside loops due to function calls in the loop body. CCIL performs almost as well as C&C. CCIL's performance is slightly worse since region boundaries are also formed right before and after the innermost loop and BCs cannot be moved across these boundaries. CCIL can help effectively break down large regions--for example in db and sor--to avoid unnecessary replays caused by buffer overflows.

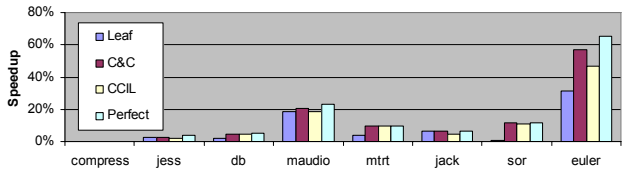

Figure 19 Speedups for leaf, C&C and CCIL.

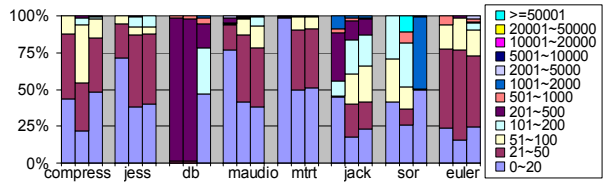

Figure 20 Regions size distributions for leaf, C&C, and CCIL (left to right) in terms of writes. Each color represents a range of the number of writes by a region.

For many applications such as compress, jess, maudio, and mtrt, the majority of regions are small ones with fewer than 100 writes in all three algorithms. If the hardware can buffer 16K writes, it can typically hold thousands of regions before a commit. Conditional region end/starts are necessary and useful for such applications. Even for other applications such as db, jack, sor, and euler, a region commit occurs, on average, after every 20 to 100 regions for most of their data points.

The leaf algorithm generates far fewer regions than the other two. However, it cannot capture some hot functions in quite a few applications like compress, db, sor, and euler. C&C generates more regions than leaf. In db and sor, some hot functions have huge multilevel loops enclosed in regions, leading to the big average region size. In sor, there are only 27 regions and a few of them generate millions of writes, leading to the big average region size. These big regions cannot be captured by the leaf algorithm. CCIL breaks down some huge multilevel loops and helps bring down the average region size.

Table 2 Total number of regions and average region size.

App	com press	jess	db	mau dio	mtrt	jack	sor	euler
Leaf	20M /28	13M /18	1.4M/ 697	29M /70	12M /11	160K/ 268	17/79	702K /300
C&C	39M /55	47M /37	1.6M/ 3104	61M /45	40M /26	1M/1 50	27/1198 4003	1M/2 31
CCIL	79M /31	49M /35	26M/1 84	73M /41	41M /25	1.2M/ 116	306K/7 80	1.4M /127

Region sizes are dependent on input sets. For jvm98 benchmarks we use the largest (100) inputs available. For Grande benchmarks we use inputs with reasonable run time. For larger inputs, better ways to break down large regions will be critical to fully explore speculative algorithms' benefit.

6.4 Discussion

Our results show that the proposed speculative algorithms can significantly increase program performance. With the hardware cost factored in, the performance improvement may not be enough to justify the use of our hybrid software/hardware model. However, we expect that transactional memory hardware (HTM) will be introduced soon to ease multithreaded programming in the Chip Multiprocessor (CMP) era. Since our proposal requires only very modest hardware extensions beyond HTM, we are optimistic that the single-thread improvements we have reported and other benefits such as improved code quality and lightweight algorithms will prove to be attractive enough to merit implementation.

Software-based speculative techniques such as check promotion and loop versioning appear in certain production JVMs and they have numerous disadvantages as described in Section 3. Our lightweight algorithms are more robust in terms of overhead, code size, and performance. If the proposed hybrid model is implemented in future systems, such systems would not need software-based speculation. Therefore, the performance potential we report from our speculative algorithms within the Jikes RVM is quite promising.

There are additional untapped performance opportunities caused by Java's precise exceptions. For example, many optimizations in Jikes are limited in dealing with PEIs. The relaxation of each such optimization and the interaction of the relaxed optimizations will lead to more performance improvement. In addition, the exceptions explicitly thrown by software and catch clauses are not considered in our work. Including them will possibly lead to greater performance benefit.

7. Related Work

7.1 Dynamic optimization

There have been many dynamic optimization systems developed since the 1990s. There are low-level native-to-native systems such as HP's Dynamo [5] and Transmeta's Code Morphing software (CMS) [10]. They translate code optimized in one ISA to another. In the translation of x86 applications to its own VLIW ISA, Transmeta's CMS can perform aggressive speculative optimization and recover to a consistent x86 excepting state with its unique hardware commit-and-rollback support. There are also high-level dynamic systems translating code in a programming language to native code. Java Virtual Machines (e.g. [3][17][25]) are typical examples.

The rePlay framework [26] relies on branch promotion to construct frames including multiple basic blocks in the hardware level. Speculative optimizations implemented in specialized hardware are then applied within the frames. Each branch is converted to an assert instruction. If the assertion fails, the frame is replayed using recovery techniques similar to ones used in current superscalar processors. In contrast with our proposal, rePlay is a purely hardware-based framework, and, as described, is limited to control-flow assertions; whereas our scheme is a hardware-software hybrid that allows more general runtime conditions (e.g. failed checks) and utilizes the compiler's greater optimization power.

ABCD [6] proposed the concept of eliminating BCs on demand for hot functions in Java. The paper gives an algorithm eliminating BCs whose index can be related to array lengths. In static compilers, researchers have proposed many techniques [15][20][29] to remove bounds checks. These techniques are heavyweight for the dynamic environment to implement. Production JVMs often make tradeoffs between compile-time overhead and the implementation's effectiveness to implement simplified versions of the aforementioned static techniques; however, the implementation details are usually not disclosed.

Modern production JVMs such as [17] started using software-based speculative techniques such as loop versioning and regioning [24] to reduce bounds check overhead. In general, such approaches suffer from execution-time as well as compile-time overhead. They only work for small loops under many assumptions and can easily lead to code bloat and become a nightmare for JVM performance tuning. For example, [17] reports that loop versioning and regioning are not one of the few optimizations that can achieve 90% of the peak performance with only 34% of the compilation time when all optimizations are used. In our scheme, much lighter algorithms can be designed and zero overhead is introduced in the commonly executed code.

There has been little research on extracting Java program performance constrained by its precise exception model. In [12], researchers propose to use software checks and recovery handlers to allow speculative code motion and significant speedups were reported on two very small kernel benchmarks due to removed precise exception constraint and the resulted loop transformations.

7.2 Hardware for guarded regions

Our proposed speculative optimization scheme requires hardware support for buffering memory and register updates and conditionally committing them or rolling them back. Such support is already available within modern processors to support control speculation, and could be extended to cover a larger region scope. Hardware transactional memory (e.g. [4][13][23][27]) or other proposals that support large-scale speculation (e.g. [2][10][26]) could be used as our underlying hardware. Accommodating large regions can also be a challenge.

While early HTM proposals [8][16][18] put a limit on the transaction size, more recent proposals have described support for larger transactions: TCC [13], VTM 0, UTM/LTM [4], and LogTM [23]. On the software side, numerous STMs have been constructed (e.g. [1][14]). The guarded region-based speculative optimization concept and the designed lightweight algorithms could be applied to Java execution on STMs.

Current approaches and proposals support hardware monitoring of conditions like predicted control-flow resolution (i.e. branches resolve the same way they were predicted) or absence of inter-thread memory access conflicts (in hardware transactional memory). To support our scheme, additional functionality would be needed to expose a more flexible interface to software and allow software and hardware to contract a richer set of runtime conditions to be monitored.

7.3 Other speculation opportunities

Static compiler and program parallelization researchers have realized that the hot execution path in a program can be executed in a speculative thread which can be committed or squashed [9]. The optimizations applied to the hot execution path can lead to significant performance improvement. This hot-path enabled aggressive optimization can be applied in our framework of a dynamic optimization system on top of hardware transactional memory or other aggressively speculative processors.

8. Conclusions

This paper proposes speculative optimization using guarded regions, a new hardware/software hybrid environment for dynamic optimization that utilizes hardware support for buffering and then committing or replaying guarded regions of code. The proposed scheme enables high performance with low compile-time overhead and ease of algorithm implementation and performance tuning. In the proposed scheme, the dynamic optimizer specifies a simple invariant—exception freedom—to the hardware, and exploits attributes of that invariant to aggressively optimize the executed program; meanwhile, the hardware must monitor the execution and trigger a guarded region replay if an exception occurs. In future work, we plan to study the benefits of specifying one or more additional runtime invariants, which can be selected from a broad range of possibilities: control flow hot paths, detection or absence of memory dependences, absence of garbage collection events, and many others. The optimization algorithms themselves can be substantially simplified, since correctness is guaranteed by the hardware-monitored invariants; this leads to marked reductions in compile-time overhead and can enable optimizations that are otherwise infeasible in a dynamic optimization environment. We further propose conditional region end/starts to ease the task of region placement. Our limit study shows that two simple optimizations--removal of null checks and bounds checks--which are impractical without the proposed support, have the potential for dramatic speedup: up to 67.7% and averaging 15.9%, with only 0.6% increase in compile time.

Acknowledgements

This research was supported in part by the National Science Foundation under grants CCR-0133437 and CCF-0429854, as well as grants and equipment donations from IBM and Intel. We would like to thank the following individuals for their input: Matthew Arnold, David Grove, Martin Hirzel, Mauricio Serrano, Ali-Reza Adl-Tabatabai, Kingsum Chow, Chris Elford, Shiliang Hu, Wei Liu, Cheng Wang, Youfeng Wu, Mark Hill, Susan Horwitz, Parameswaran Ramanathan, Mike Schulte, Jim Smith, and fellow PHARM members. We thank the anonymous VEE reviewers for their extremely helpful comments and feedback.

References

[1] A.-R. Adl-Tabatabai et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In PLDI 2006.

[2] H. Akkary et al. Checkpoint processing and recovery. In Micro36 2003.

[3] B. Alpern et al. The Jalapeno Virtual Machine. IBM Systems Journal, 39(1):211-221, 2000.

[4] C. S. Ananian et al. Unbounded Transactional Memory. In HPCA2005.

[5] V. Bala et al. Dynamo: A Transparent Dynamic Optimization system. In PLDI 2000.

[6] R. Bodik et al. ABCD: Eliminating Array Bounds Checks on Demand. In PLDI 2000.

[7] M. Burke et al. The Jalapeno Dynamic Optimizaing Compiler for Java. In ACM Java Grande Conference 1999.

[8] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. ACM Transactions on Computer Systems, 1988.

[9] L. Chen & Y. Wu. Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation. In ICPP 2003

[10] J. Dehnert et al. The Transmeta Code Morphing Software. In CGO'03.

[11] S. J. Fink & Q. Feng. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In CGO'03.

[12] M. Gupta, J.-D. Choi, and M. Hind. Optimizing Java Programs in the Presence of Exceptions. In ECOOP 2000.

[13] L. Hammond et al. Transactional Memory Coherence and Consistency. In ISCA 2004.

[14] T. Harris et al. Optimizing Memory Transactions. In PLDI 2006.

[15] W. H. Harrison. Compiler Analysis for the Value Ranges of Variables. IEEE Transactions on Software Engineering 1977.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In ISCA 1993.

[17] K. Ishizaki et al. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler. In OOPSLA 2003.

[18] T. Knight. An Architecture for Mostly Functional Languages. In ACM Conference on LISP and Functional Programming 1986.

[19] T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification (2nd Edition). 1999. Addison-Wesley London.

[20] V. Markstein et al. Optimization of Range Checking. In Proceedings of Symposium on Compiler Optimization 1982.

[21] J. Martinez & J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In ASPLOS'02.

[22] A. McDonald et al. Architectural Semantics for Practical Transactional Memory. In ISCA 2006.

[23] K. Moore et al. LogTM: Log-based Transactional Memory. In HPCA 2006.

[24] J. Moreira et al. From Flop to Megaflops: Java for Technical Computing. In TOPLAS 2000.

[25] M. Paleczny, et al. The Java HotSpot Server Compiler. In JVM 2001.

[26] S. J. Patel and S. S. Lumetta. rePlay: A Hardware Framework for Dynamic Optimization. In MICRO 2000.

[27] R. Rajwar et al. Virtualizing Transactional Memory. In ISCA 2005.

[28] M. Spezialetti and R. Gupta. Loop Monotonic Statements. IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995.

[29] N. Suzuki and K. Ishihata. Implementation of an Array Bound Checker. In POPL 1977.

[30] Java Grande. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[31] The Performance Counter Library (PCL). http://www.fz-juelich.de/zam/PCL/.

[32] SPECJVM98 1998. http://www.spec.org/jvm98