Lazy Interprocedural Analysis for Dynamic Loop Parallelization

Jisheng Zhao, Christopher Kirkham, and Ian Rogers

The University of Manchester, School of Computer Science,
Oxford Road, Manchester, UK
{jisheng.zhao,christopher.kirkham,
ian.rogers}@manchester.ac.uk
http://www.cs.manchester.ac.uk/apt

Abstract. Dynamic compilation is becoming a dominant compilation technique. Runtime compilation has to avoid slow compile times by targeting optimizations to areas where it has a performance impact. For parallelization optimizations this can lead to not exposing opportunities for parallelization. To enable fuller optimization we present a simple interprocedural analysis. Our analysis and parallelization phases are performed as part of the Jikes RVM. Our approach succeeds in finding coarser grain loops and increased performance in a number of benchmark kernels on a research chip multi-processor architecture.

1 Introduction

Parallel computers are now becoming ubiquitous. The JAMAICA Chip Multi-Processor (CMP) architecture [1] exposes a large number of contexts capable of running fine-grained threads, threads created, scheduled and deleted with a low cost. In order to expose these threads parallel languages, libraries and parallelizing compilers exist. The approach we present concentrates on exposing fine-grain threads through automatic parallelizing compilation focused on loops.

Dynamic compilation is a technique widely used for Java and C# applications. In this work we look at modifying the Jikes RVM [2], a Java Virtual Machine (JVM) responsible for managing the runtime, garbage collection and compilation environment. In such a dynamic compilation environment, heavily executed code is optimized using an optimizing compiler. In our environment we've extended the optimizing compiler to also perform dynamic parallelization. In general, the cost for the analysis and performing the parallelization optimization is high. A balance is required, bringing about the need for safe parallelization whilst avoiding the cost of a full optimizing compile.

Increasing the granularity of parallel tasks can reduce the overhead of parallel thread creation and, hence, benefit performance. It is important to parallelize outer loops that expose the largest granularity parallel tasks. Such loops often contain method calls, especially in scientific computing codes. If the parallel compiler can optimize loops in the presence of method calls coarser-grain loops

can be created whilst avoiding expensive analysis of the program as a larger hot region.

In this paper, we present an approach for runtime interprocedural analysis used in the Jikes RVM's dynamic compilation system to enhance automatic loop level parallelization. We present an analysis of the resultant system on the JAMAICA architecture using a modified version of the Jikes RVM called the JaVM. This system has a modified compiler and runtime system that support the JAMAICA architecture and its fine-grain threading mechanisms.

The rest of the paper is organised as follows: in section 2, we present more background material about the requirements for our adaptive parallelization and interprocedural analysis system. In section 3, we present the interprocedural analysis mechanism. In section 4, we discuss some issues in implementing interprocedural analysis in the JaVM dynamic compilation system. We analyse the performance of our approach in section 5. We conclude this paper in section 6.

2 Dynamic Parallelization in the JaVM

2.1 The JAMAICA CMP

The JAMAICA CMP [1] is an architectural development platform, being built around a configurable simulator. JAMAICA has similarities to other CMP architectures, such as Stanford's multi-core Hydra [3] or the multi-core and multi-threaded Sun Niagara architecture[4]. The JAMAICA architecture differs by having light-weight hardware support for thread creation, scheduling and distribution [5]. A processor signals to a thread distribution network that it has a context available, another processor can request this context and ship work to it by a thread shipping procedure call. The same mechanism is used to distribute loop iterations among different processors.

2.2 JaVM and Dynamic Parallelization

The JaVM is a port to the JAMAICA architecture of the Jikes RVM. It contains a runtime service system (thread scheduling, memory management) that utilises the JAMAICA CMP architecture, a dynamic compiler and an adaptive recompilation system.

There are two compilers in the Jikes RVM: a baseline compiler that maps Java bytecode to JAMAICA machine code directly and a optimizing compiler [6] that translates to machine code via an *intermediate representation* (IR) and a number of compiler optimization phases. The automatically parallelizing compiler has been built on this optimizing compiler; it performs dependence analysis on high level IR (HIR) code, and adds thread shipping and scheduling code to the machine level IR (MIR) code [7].

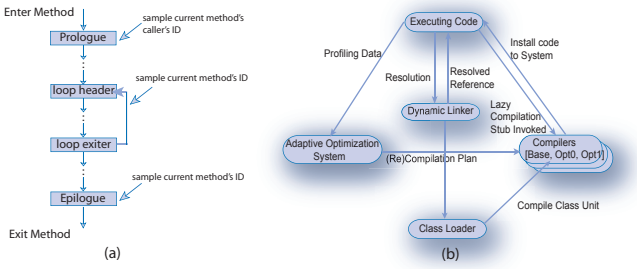


Fig. 1. Sampling mechanism and Jikes RVM adaptive optimization system

2.3 Adaptive Parallelization System

The adaptive optimization system [8] shown in figure 1(b) uses a runtime profiling mechanism to find hot methods that are either called frequently or have heavily executed loops. The compiler inserts yield points into the Java methods[1]. Each yield point checks whether the current thread needs to yield[2], and performs runtime sampling for the adaptive system when it does. It captures the relevant method's ID[3] and records it in a sampling array. Those methods whose IDs exist in the sampling array with high frequency are *hot*. There are three positions where the yield points are inserted (shown in figure 1(a)):

- Method Prologue: executed on entry to a method.
- Loop Backedge: executed when following the backedge of a loop.
- Method Epilogue: executed upon leaving a method.

Hot methods are recompiled with more optimizations to improve their performance. In JaVM all of the application methods would be compiled initially by the baseline compiler. The adaptive system performs a cost benefit analysis of how and whether to recompile hot methods, at which point parallelization can also occur. The adaptive parallelization system (APS) is therefore a combination of two major components: a parallel compiler working dynamically, used to create parallel code within methods, and the adaptive optimization system, used to drive the adaptive recompilation process tailored toward parallelization.

2.4 Parallelizing Compiler Architecture

The parallel compiler is embedded into JaVM's optimizing compiler. Shown in figure 2, the optimizing compiler comprises a series of compilation phases.

[1] The compiler does not insert any yield points into native methods or mission critical methods, i.e. those methods used for runtime scheduling and memory management.

[2] There's an internal counter; the count value will be incremented every time the yield point function is called. If the count value exceeds a predefined value, the current thread should yield and the count value should be set back to 0. Based on this mechanism, the Java thread could yield after an imprecise time interval.

[3] The method ID is an integer value assigned to a method when it is compiled.

The parallel compiler comprises several compilation phases that perform the data dependence test[9], parallel loop annotation at HIR level and parallel code generation at MIR level.

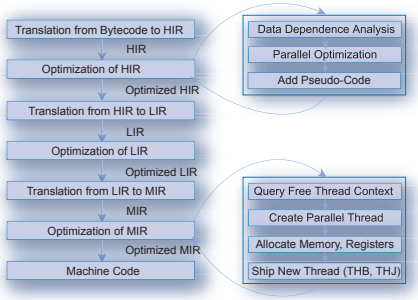
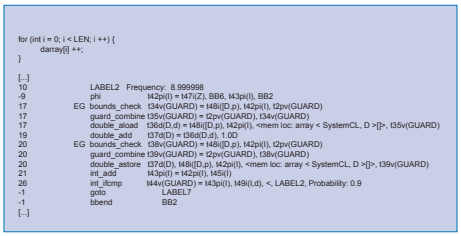


Fig. 2. JaVM parallel compiler

Figure 3 shows an example simple Doall loop and its HIR code before being annotated by the parallel compiler, it is then passed by through the analysis and annotation phase. Figure 4 shows the annotated HIR code (pseudo-codes are added to annotate the parallel loop and its loop invariants, and a loop versioning optimization is performed here to eliminate runtime exceptions [7, 10]). Figure 5 shows the MIR code, when the annotated loop passes the parallel code expanding phase at MIR level, part (a) is the code for parallel thread creation and part (b) is the code run by each parallel thread.



```
for (int i = 0; i < LEN; i ++) {
        darray[i] ++;
}

[...]
10                LABEL2   Frequency:  8.999998
-9                phi              t42pi(I) = t47i(Z), BB8, t43pi(I), BB2
17             EG bounds_check     t34v(GUARD) = t48i([D,p), t42pi(I), t2pv(GUARD)
17                guard_combine t35v(GUARD) = t2pv(GUARD), t34v(GUARD)
17                double_aload    t36d(D,d) = t48i([D,p), t42pi(I), <mem loc: array < SystemCL, D >[]>, t35v(GUARD)
19                double_add      t37d(D) = t36d(D,d), 1.0D
20             EG bounds_check     t38v(GUARD) = t48i([D,p), t42pi(I), t2pv(GUARD)
20                guard_combine t39v(GUARD) = t2pv(GUARD), t38v(GUARD)
20                double_astore   t37d(D), t48i([D,p), t42pi(I), <mem loc: array < SystemCL, D >[]>, t39v(GUARD)
21                int_add         t43pi(I) = t42pi(I), t45i(I)
26                int_ifcmp       t44v(GUARD) = t43pi(I), t49i(I,d), <, LABEL2, Probability: 0.9
-1                goto                LABEL7
-1                bbend               BB2
[...]
```

Fig. 3. HIR prior to parallelization

2.5 Interprocedural Analysis

The automatic parallelizing compiler has to process method calls within loops and determine whether they can be allowed or not. To avoid this problem, method inlining is used to substitute a call with the body of the method it would call [6]. Because of the adaptive optimization system's analysis, not all methods will be inlined. Typical methods that aren't inlined are methods that contain too many instructions or complex control flow.

In general a loop body that contains a method call cannot be parallelized because the method may have a hazard inducing side effect. There are three major types of side effect inside methods:

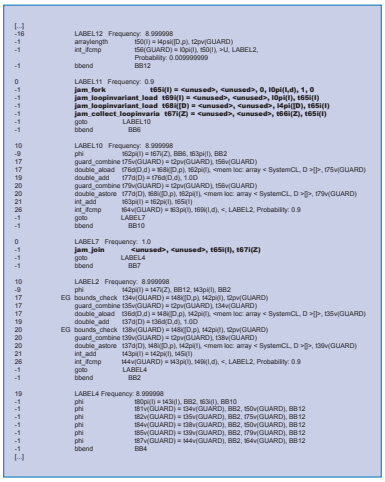


Fig. 4. HIR following parallelization

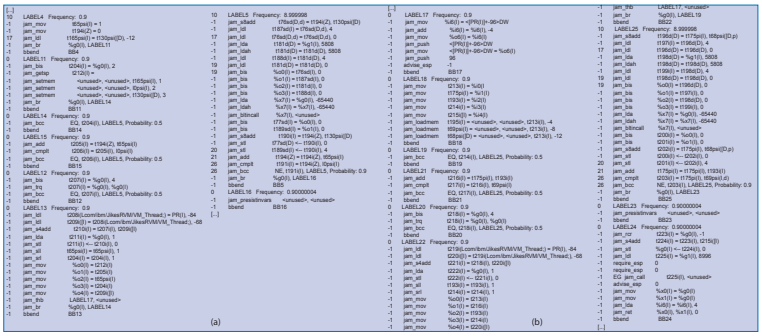


Fig. 5. After expansion of parallel instructions in MIR

– Data Dependency:
  - If methods called within a loop body contain write operations to the same address a *write after write* (WAW) or *output* data dependency will exist and means the loop bodies can't be executed out-of-order.
  - Similarly if the method called contains a read from a memory address written during the loop body a *write after read* (WAR) or *anti* data dependency will exist.
  - Finally, if the method called writes data to a memory address and that address will be read by a following loop iteration, a *read after write* (RAW) or *true/flow* data dependency exists.
– Exceptions: the Java language uses the exception mechanism to handle exceptional situations at runtime. These include Java runtime exceptions and user defined exceptions. An exception can break the program's normal control flow and trap to special exception handlers. Potentially all methods called within a loop can throw an exception, and this may make the loop terminate early.

– Synchronization: if there's a use of locks in the code considered for parallelization, parallelizing a loop that calls code with locks in it can result in the order the locks are acquired changing. This raises the potential of deadlocking the loop, by a later loop iteration acquiring a lock required by an earlier loop. We avoid parallelizing code that contains locks.

For those loops which don't fail dependence analysis but contain loop method calls, we need to perform interprocedural analysis to determine whether or not the loop could be parallelized. For this reason we need a low cost and efficient interprocedural analysis mechanism.

3 Parallelization with Lazy Interprocedural Analysis

The adaptive system described in section 2.3 focuses effort on hot methods in a way that's suitable for recompilation of hot spots. For interprocedural analysis, again the expense is only justified for hot methods, however, interprocedural analysis may expose loop level parallelism in loops surrounding hot methods that aren't themselves as hot. Parallelism exposed in this way will be of a coarser granularity compared to loops within the hot method. Due to the way sampling occurs in the Jikes RVM, we found inner methods were commonly recompiled before outer methods. Our approach tackles this problem in two phases:

– firstly, we produce guarded parallel and non-parallel outer loops in all methods compiled by the optimizing compiler containing calls that could be potentially run in parallel.
– secondly, we perform interprocedural and class hierarchy analysis when classes are loaded, and when optimizing classes, to determine when it is safe to use parallel outer loops containing method calls.

Here's a simple example, func_A contains a potentially parallel loop that contains no loop carried data dependencies, except two loop method calls: func_B and func_C. The adaptive system captures func_A as a hot method. The parallel compiler performs the data dependence analysis on this method's loop and finds its suitable to be executed in parallel if the method calls don't expose data dependencies. The compiler checks whether the two method calls within the loop are known not to contain any side effects. Because there's not enough information currently the parallel compiler compiles func_A with two loop versions, and uses the non-parallelized loop as the initial version. Later, when the two callees are chosen for compilation and analysis by the adaptive system, they are annotated to contain information regarding what data hazards they potentially expose. Once this information is known, the parallelization system determines whether the loop in func_A can be run in parallel, and if it can, the loop version is switched. Because of later dynamic class loading, and virtual method dispatch, we may need to switch the loop version back to the non-parallel version. However, again we may prove later, following analysis, that the loop in func_A is safe to execute in parallel and we switch the loop to the parallel version.

To aid this mechanism it is useful to maintain a partial call-graph, which is complicated by dynamic class loading. A limitation of our current call-graph analysis is that it only considers static methods, or virtual methods of classes that can't have sub-classes (either through use of the final keyword, or by making their constructors private). Whilst this has been suitable in benchmark code, greater performance may be achievable in general purpose code by maintaining more accurate call-graph information in the face of dynamic class loading. Such analysis is performed in [11].

On stack replacement[12] provides a means to replace running code, for example if method inlining proved unsafe following a new class being loaded. Our approach differs, as when we need to replace code we don't first recompile it. In general more speculative optimizations are only applied to hot methods compiled at higher compilation levels. As we are creating parallel code early, at lower compilation levels, we believe our technique more closely suits our requirement to produce greater parallelism.

3.1 Method Selection

The JVM provides 4 mechanisms for method call, in general method calls are either to a known location (such as static methods) or to methods determined by the type of the object the method is called upon (such as virtual or interface methods) [13]. Our analysis must determine whether a method called is safe for parallelization, for this we must ensure the following constraints:

For virtual methods:

- It should only have load operations on initialised final fields and static fields whose data type is primitive, e.g. int, long, double.
- The input parameters should not be object references.
- There are no instructions that might throw a Java runtime exception.
- There are no monitor instructions for synchronization.
- For the callees of this method:
  - If the callee is a virtual method and its object reference is same as the caller, it should have the same constraints as the caller.
  - If the callee is a static method, the callee should not have any *putstatic* operation and its *getstatic* operations should only work on primitive types.

For static methods:

- The static method can not perform any *putstatic* operations.
- All of the *getstatic* operations only work on primitive types.
- The input parameters should not be object references.
- There are no instructions that might throw a Java runtime exception.
- There are no monitor instructions for synchronization.
- If this method contains static method calls, the callees should not have *putstatic* operations and their *getstatic* operations should only work on primitive types.

These constraints prohibit methods called from the loop from performing any memory load/store operations on the class fields within the same class as its caller method; the VM can make sure that there's no data dependency related to the method called in the loop. As initialised static final fields whose types are primitive can be treated as constant values, we allow for load operation on such fields.

The *putstatic* instruction stores a value to a static class field, so if the loop contains *putstatic* operations, that will generate output dependencies. The *getstatic* instruction loads a value from a static class field; if the loop contains any *getstatic* operations, that may imply that the loop may do some load operation on the class object and generate a flow or anti dependencies.

3.2 Call Graph Analysis

Method calls and the runtime call-graph are used to determine the relationship among the hot methods. There are three hash tables used to maintain this relationship and method states.

- Callees map: the keys of the table are method object references[4] and the values are the set of methods called by the key method.
- Callers map: the keys of the table are method object references and the values are the set of methods that call the key method.
- Loop Callees map: the keys of the table are method object references and the values are the set of methods called by the loop in the key method. If there are more loops than one, we should allocate more than one method set to correspond to the different loops.

We extend the information known about a method by introducing new summary information for each method:

- *hasPSOperation*: the method contains a putstatic operation or a putfield operation to a field visible outside of the class.
- *hasGSOperation*: the method contains a getstatic operation that works on an object reference or a getfield operation that works on an object reference visible outside of the class.
- *hasLSOperation*: the method contains memory load/store operations to fields within the class that aren't exposed outside of the class. The method is virtual.
- *hasException*: the method contains instructions which might throw a Java runtime exception.
- *hasSynchronization*: the method contains synchronized (monitor) segments.
- *hasObjParam*: the method's input parameters contain an object reference.
- *unclear*: the method has not been analysed currently, or there are some callees of this method which have not been analysed.

An example of the data organisation is shown in figure 6 and 7. The *Loop Callees Map* is a subset of *Callees Map*, it records all the loop method calls related to the loop.

[4] In JaVM, the method object reference is an object reference to a VM_Method object.

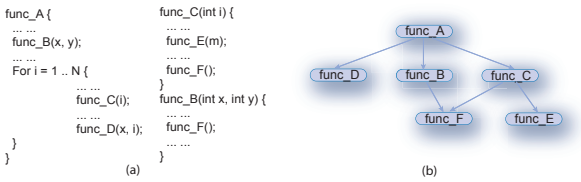


Fig. 6. Example of runtime call graph

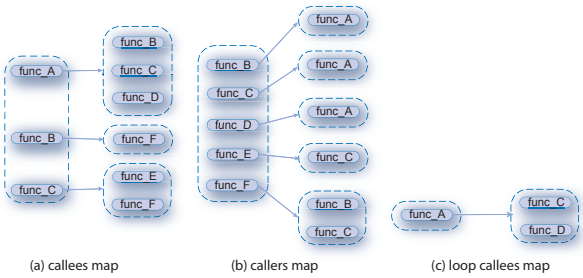


Fig. 7. Example showing runtime call graph maps

Every method's summary should be a combination of the state shown above. The caller method should check its callee methods' states and update its own states, or when a callee method finishes its analysis, it should notify its caller method and make its caller method update states. The rules for updating states are listed here:

- If the callee has state *hasPSOperation*, the caller should get *hasPSOperation* state.
- If the callee has state *hasGSOperation*, the caller should get *hasGSOperation* state.
- If the callee has state *hasException*, the caller should get *hasException* state.
- If the callee has state *hasSynchronization*, the caller should get *hasSynchronization* state.
- If the callee has state *hasLSOperation* and the callee and caller are within same class, the caller should get *hasLSOperation* state.

3.3 Implementation Detail

A detailed explanation of the interprocedural analysis for a hot method event is described below. A similar process is performed when a class is loaded.

1. Check each instruction, and set the method's state as listed in section 3.2.
2. For each call instruction:
   - set the current method and the call target method in the callee map.
   - set the current method and the call target method in the caller map.
   - if the call target method (callee) has been analysed (it does not have *unclear* state), collect its states and update the current method states based on the rules listed in section 3.2, and if not, set the *unclear* state to current method.

3. When the instruction is finished, and it does not contains any states listed in section 3.2. That means that this method could be part of a parallel loop. Then do following steps:
   (a) get the callers of this method from the callers map.
   (b) for each caller, check the loop callees map to find out whether this method has been called in a caller's loop that might be parallelized but which depends on some loop method calls.
   (c) remove this method's reference from the loop method call set corresponding to the caller's loop, which means that the caller's loop doesn't depend on this method.
   (d) of the loop method call set is empty, that means that the corresponding loop can be parallelized. Switch the caller method's execution path to the parallel version of loop.
4. If the current method contains some loops, the parallel compiler should perform data dependence test on those loops.
5. If one loop could be parallelized but contains method calls check the loop method calls' states:
   – If all of the loop method calls don't contain any states listed in section 3.2, then the loop should be compiled as a parallel loop.
   – If some of the loop method calls' states contain *unclear*, the compiler can not make decision at the current time. The loop has to be compiled with two versions and the execution path should be set to the non-parallel version initially.
   – If any of the loop method calls' states contain the state listed in section 3.2 except *unclear*, that means there are potential side effects in the loop and the loop should not be parallelized.

4 Implementation Considerations and Extension

4.1 Data Dependence Test

With dynamic compilation the trade-off between compilation time and improved performance is important. For this reason we use the GCD test [14] to determine whether parallelization is safe. However, this test is simple and may be overly conservative [15].

JikseRVM provides an extended Array-SSA[16] form that can be used to analyse the data dependence between the load/store operations on array elements. An important property of this form is that it uses Java's strong typing information to remove many potential false dependencies.

Data dependency testing starts from the outer-most loop and proceeds to the inner-most one; if a loop passes the test, then it should be annotated as a potentially parallel loop.

For loop method calls, their input parameters should be treated as memory load operations and their return values correspond to memory store operations. Because of the constraints listed in section 3.1, the method should not contain any potential memory load/store operation related to other memory store/load operations in the loop body.

4.2 Library Methods

Library methods differ from methods dynamic loaded in that we can know ahead of time information about what dependencies exist. This information can be placed explicitly in the compiler or through annotations to the library. For the pure mathematical operations of *java.lang.Math* we therefore avoid any analysis as we know they contain no side effects.

4.3 Exception Elimination

In section 2.5 we described that loops containing method calls that throw exceptions couldn't be parallelized. During compilation exceptions are eliminated and therefore we can potentially determine that loops can be parallelized. We classify methods containing exceptions into two categories:

- Explicit Exceptions: this is a method that when called may explicitly throw an exception.
- Potential Exceptions: these exceptions relate to runtime checking operations. These are array bound, null object reference, object array store and class cast checks.

Currently, we just check all such exceptions listed above, but in practice some of them are redundant. In the example shown in figure 8, the caller func_A creates an array whose length is 10, and passes this array reference to callee func_B. In func_B, there are no array element load/store operations which exceed the array length. So the array bound check should be redundant in this call. To implement this function, the runtime call graph would need to be extended so it could store more runtime information for the optimizer.

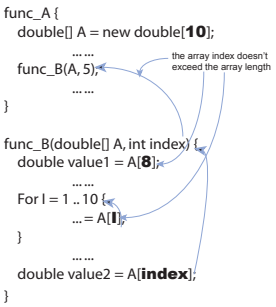


Fig. 8. Interprocedural Array Bound Check Problem

4.4 Switching Loop Versions

In JaVM, each of the Java methods has its own code array holding the compiled machine code; the code array is referred to by virtual method tables and also

an index in the JTOC[5]. A jump controls which version of the loop should be executed, as shown in figure 9. The jump is overwritten to switch between the loop versions. As the jump is outside of the loop, a thread that enters the loop can't witness a change to the jump. To enable new executions of the code to witness the alteration to the jump we flush the data and instruction caches at a point when no thread can be witness a glitch in the jump instruction changing. To support switching the jump instruction, the compiler records the machine code offset of the jump during the optimizing compilation.

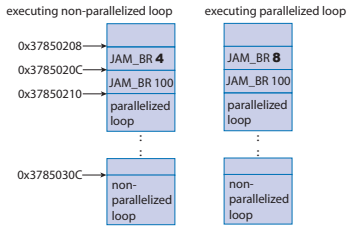


Fig. 9. Loop Version Switching

4.5 Reducing the Input Parameter Constraint

Section 3.1 lists all of the constraints for method selection, including that the loop method call's input parameters should not be object references. The reason for adding this constraint is to prohibit potential memory operations on the input object which may have side effects. If the input object is a simple object whose data fields are all primitive type, with no static data fields and all of whose methods obey the constraints in section 3.1, then there should not be any data hazards. The compiler needs to make sure there are no data hazards related to the loop method call's input object[6] in the loop body's scope.

There are two issues that need to be addressed: how to check the object is suitably simple, and remove the *NULL_CHECK* for the input object reference. To check the object is simple, we need to check all of its super classes recursively, to make sure that all of the class fields are primitive. Currently, to avoid problems caused by dynamic class loading we add a constraint that the input object's class should not have any subclasses.

The methods whose input parameters have object reference type must have *NULL_CHECK* instructions when the object reference is first accessed. We extended the compiler analysis phase to consider that if the caller method passes a non-null reference to a callee, then that callee's *NULL_CHECK* operation for this object is not necessary. As the callee may be recompiled earlier than its caller, we have to add a new state for the callee method: *hasInObjNullCheck*

[5] Java Table of Content - a table holding static information from classes.

[6] In this scenario, the input object should be treated as both of memory load/store operation.

which means that all of the *NULL_CHECK* operations are related to the input object references, so this callee method could be part of parallel loops when the compiler can make sure that all of its input objects are non-null.

5 Evaluation

The test result was obtained using an architectural simulator of the JAMAICA chip multi-processor architecture. The configurations used ranged from having 2 to 16 processors, all having one thread context per processor. Our results are normalized against the execution time of the serial program, which can execute on just one processor. Three of the benchmark kernels chosen were taken from jBYTEMark[17], with the MoldynTest coming from the serial Java Grande Forum Benchmark Suite[18].

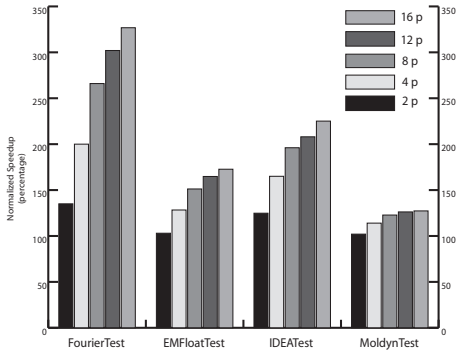


Fig. 10. Automatic parallelization with interprocedural analysis on a range of benchmark kernels

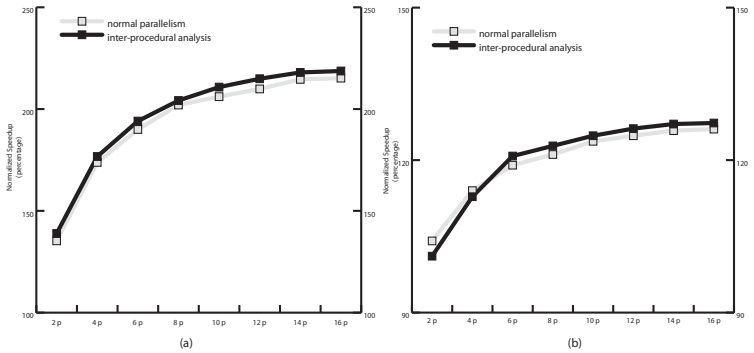


Fig. 11. Comparison of automatic parallelization of the MoldynTest

The results show that JaVM's parallelization optimization is able to create parallel code and improve performance. All the results show that performance increases with a greater number of parallel processors. In figure 10 a maximum speedup of 3.25 times was achieved. However, the performance improvement gradually decreases as contention for the memory system becomes a problem. The performance also didn't scale fully with the number of processors as the parallel region was only a small section of the running benchmark kernel and virtual machine code.

Figure 11(a) shows that the interprocedural analysis is able to get a greater speedup on the MoldynTest than just regular parallelization. The analysis phases cause a small increase in compilation cost, which in figure 11(b) can be shown to cause a slight deterioration in performance for the interprocedural analysis when compared to just automatic parallelization for 2 and 4 processor configurations.

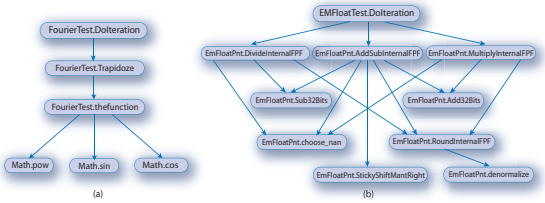


Fig. 12. Call Graph for FourierTest and EMFloatTest

Figure 12 shows a call graph for two of the benchmarks. The benchmarks demonstrated the problem for automatic dynamic parallelization of the adaptive compilation system. For example, *FourierTest* contained a parallelizable outer loop in the method *DoIteration* which was inhibited by method calls to *thefunction* and *TrapezoidIntegrate*. Following analysis of these methods the interprocedural system was able to enable the parallel loop in *DoIteration*. Similarly for *EMFloatTest*, the parallel loop in *DoIteration* was created during compilation of the 4th method, with the analysis enabling parallelization during compilation of the 8th method.

6 Conclusion

This paper addresses problems with dynamic parallelization for CMP environments. It's presented efficient and low cost analysis approaches that can generate parallel regions without the cost of recompilation. The effectiveness of these approaches was shown by parallelizing previously serial Java programs, in particular a number of programs containing scientific kernels, and demonstrating a speedup beyond that achieved by just automatic parallelization.

Currently, this analysis focuses on checking for potential side effects between the caller and callee methods. With many memory operations this analysis may be conservative. In future work, we plan to extend the runtime data structure to store a greater amount of information to yet further expand automatic parallelization at runtime.

References

1. : The Jamaica project. http://www.cs.manchester.ac.uk/apt/projects/jamaica (2006)
2. IBM: Jikes^{TM}Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net/ (2006)
3. Hammond, L., Hubbard, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukotun, K.: The Stanford Hydra CMP. IEEE Micro (2000) 71–84
4. Nagarajayya, N.: Improving application efficiency through chip multi-threading. Sun developers forum (2005)
5. Wright, G.: A single-chip multiprocessor architecture with hardware thread support. PhD thesis, The University of Manchester (2001)
6. Burke, M., Choi, J., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M., Sreedhar, V., Srinivasan, H., Whaley, J.: The Jalapeeño dynamic optimizing compiler for Java. In: Proceedings ACM 1999 Java Grande Conference, San Francisco, CA, United States, ACM (1999) 129–141
7. Zhao, J., Rogers, I., Kirkham, C., Watson, I.: Loop parallelisation for the JikesRVM. In: Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2005. (2005)
8. Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: Adaptive optimization in the Jalapeño JVM. In: Conference on Object-Oriented. (2000) 47–65
9. Wolfe, M.J.: High performance compilers for parallel computing. Addison-Wesley, Redwood City, CA (1996)
10. Zhao, J., Rogers, I., Kirkham, C.: A system for runtime loop optimisation in the JikesRVM. In: Postgraduate Research in Electronics, Photonics and Communication (PREP 2005). (2005)
11. Qian, F., Hendren, L.J.: Towards dynamic interprocedural analysis in JVMs. In: Virtual Machine Research and Technology Symposium. (2004) 139–150
12. Fink, S., Qian, F.: Design, implementation and evaluation of adaptive recompilation with on-stack replacement (2003)
13. Sun Microsystems Inc.: The Java Virtual Machine Specification. 1.0 beta edn. (1995)
14. Banerjee, U.: Loop Transformations for Restructuring Compilers, The Foundations. Kluwer Academic Publishers, Boston (1994)
15. Petersen, P.M., Padua, D.A.: Static and dynamic evaluation of data dependence analysis techniques:. IEEE Transactions on Parallel and Distributed Systems **7**(11) (1996) 1121–1132
16. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Symposium on Principles of Programming Languages. (1998) 107–120
17. Byte.com: The jBYTEMark. http://www.byte.com (2006)
18. Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A methodology for benchmarking java grande applications. In: JAVA '99: Proceedings of the ACM 1999 conference on Java Grande, New York, NY, USA, ACM Press (1999) 81–88