Dynamic Path Profile Aided Recompilation in a JAVA Just-In-Time Compiler*

R. Vinodh Kumar[1], B. Lakshmi Narayanan[2], and R. Govindarajan[3]

[1] Cisco Systems, Bangalore, India
vinodh@cisco.com
[2] School of Computer Science and Engineering, College of Engineering-Guindy
Anna University, Chennai 600025, India
blnarayanan@hotmail.com
[3] Department of Computer Science and Automation and Supercomputer Education and Research Centre, Indian Institute of Science
Bangalore 560 012, India
govind@csa.iisc.ernet.in

Abstract. Just-in-Time (JIT) compilers for Java can be augmented by making use of runtime profile information to produce better quality code and hence achieve higher performance. In a JIT compilation environment, the profile information obtained can be readily exploited in the same run to aid recompilation and optimization of frequently executed (hot) methods. This paper discusses a low overhead path profiling scheme for dynamically profiling JIT produced native code. The profile information is used in recompilation during a subsequent invocation of the hot method. During recompilation tree regions along the hot paths are enlarged and instruction scheduling at the superblock level is performed. We have used the open source LaTTe JIT compiler framework for our implementation. Our results on a SPARC platform for SPEC JVM98 benchmarks indicate that (i) there is a significant reduction in the number of tree regions along the hot paths, and (ii) profile aided recompilation in LaTTe achieves performance comparable to that of adaptive LaTTe in spite of retranslation and profiling overheads.

1 Introduction

A Java JIT compiler is a component of the Java Virtual Machine (JVM) [20], which translates Java's bytecode methods into native code prior to execution, so that the Java program under execution runs as a real executable. Since the

* This work was done when the first author was a graduate student at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore. It was extended when the second author was a Summer Research Fellow of Jawaharlal Nehru Centre for Advanced Scientific Research, Bangalore, at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore. The last author acknowledges the research funding received from the Department of Science and Technology, India, which partly supported this work.

S. Sahni et al. (Eds.) HiPC 2002, LNCS 2552, pp. 495–506, 2002.
© Springer-Verlag Berlin Heidelberg 2002

machine code is *cached* in the JVM, the translated native code is used for subsequent executions of the method. The execution of the translated code avoids the inefficiencies associated with the interpretation of bytecode. However, such a translation incurs runtime overheads and hence it cannot apply aggressive compiler optimizations to obtain efficient code. To overcome this, adaptive compilation techniques, which apply aggressive compiler optimizations only on frequently executed methods, have been proposed in the recent past [2, 11, 19]. Such adaptive compilation facilitates the exploitation of runtime program behavior through profile information for better performance.

Programs typically spend most of their time in a small set of paths and optimization along these paths definitely yields performance benefits. In the static mode of compilation, path profiles from preparatory runs of the program are used for off-line re-compilation and optimization of the program along its frequently executed paths. Several profiling methods [5, 6, 7] have been proposed to reduce the time incurred in obtaining the profile information. In a dynamic compilation environment, programs can be monitored for some time and the parts of program which are expected to run frequently for the remaining part of the execution can be optimized. This form of adaptive compilation of programs exists in modern JIT compilers such as HotSpot compiler from *Sun* [14], *Intel* Vtune JIT compiler for Java [1], and LaTTe [19]. These compilers retranslate and optimize methods that are invoked more frequently (hot methods).

In our work, we plan to aid the retranslation of the hot methods in a Java JIT compiler using intra-procedural (i.e., intra-method) path profiling information gathered from a low overhead path profiling strategy so that the methods can be better optimized for execution along the hot paths. The technique adopted for profiling must have low (profile) instrumentation overheads and low profiling overheads, since these constitute a part of the running time of the program. As a consequence, sophisticated low profile overhead algorithms such as the Ball and Larus' efficient path profiling algorithm [6] or the efficient edge profiling algorithm [5], which take significant time to identify profile points in the program, cannot be used in dynamic compilation.

We use a profiling strategy called *bit tracing* [5] for generating the path profiles dynamically. Using the profile information, we enlarge tree regions, which are the units of optimization in LaTTe, along the hot path. For this purpose, we clone basic blocks along the hot path which have more than one incoming edge in the control flow graph to ensure that all the basic blocks along the hot path are within a single region. This allows the optimizations to be more effective on the hot paths in the program. The profile information is then further used to perform instruction scheduling on superblocks [10] which are present along the hot paths. Thus the major contributions of this paper are: (i) implementing an efficient profile-assisted aggressive recompilation method for hot paths in an adaptive JIT compiler, (ii) expanding regions to include all basic blocks of a hot path, and (iii) superblock instruction scheduling in a JIT compiler framework. Our initial results show that region expansion along the hot paths significantly reduces the number of regions in a method, by a factor of 4 to 5, in all SPEC

JVM98 benchmarks. This facilitates aggressive optimization along the hot path. Our experiments with superblock instruction scheduling reveal that profile aided recompilation in LaTTe achieves performance comparable to that of adaptive LaTTe in spite of the retranslation and profiling overheads. This suggests that if retranslation overheads, when retranslating again from bytecode to native code, can be mitigated, profile assisted aggressive compilation methods could yield further performance improvements.

Section 2 presents the necessary background on LaTTe. In Section 3, we discuss the path profiling technique used in our implementation. Section 4 deals with the recompilation strategy. Section 5 focusses on region expansion and superblock scheduling performed in our implementation. In Section 6 we present our performance results. Section 7 discusses related work. In Section 8 we provide concluding remarks.

2 LaTTe JIT Compiler

LaTTe is a Java virtual machine [19] that includes a JIT compiler targeted at RISC machines, specifically the Ultra SPARC processor [17]. The JIT compiler generates good quality code through a mapping of the Java Stack to registers, incurring very little overhead. It also performs some traditional optimizations such as common subexpression elimination and loop invariant code motion [19]. In LaTTe, the basic unit of optimization is a tree region (or simply, region). The CFG of pseudo SPARC code is partitioned into regions which are single entry, multiple-exit sub-graphs shaped like trees. Tree regions start at the beginning of a method or at other join points and end at start of the other regions.

In LaTTe, translation of bytecode to native code is done in 4 phases. The details of these phases can be found in [19].

BYTECODE_ANALYSIS: LaTTe identifies all control join points and subroutines in the bytecode via a depth first traversal.

CFG_GEN: The bytecode is translated into a control flow graph (CFG) of pseudo SPARC instructions with symbolic registers. If speculative virtual method inlining is enabled, it is done during this phase.

OPT: In the third stage, LaTTe performs region-based traditional optimizations, along with region-based register allocation.

CODE_GEN: In the final phase, the CFG is converted into SPARC code, which is allocated in the heap area of the JVM and is made ready for execution.

Adaptive retranslation in a JIT compiler is used to perform the costly runtime optimizations selectively on hot methods based on the program behavior. In the adaptive version of LaTTe, methods are selected for optimizations based on method run counts [15]. This is achieved as follows. When a method is called for the first time, it is translated without any optimization and method inlining. Each method has an associated method run counter which is initially loaded with a threshold value and decremented each time the method is invoked. When the method count becomes zero, i.e., when the number of times this method is

invoked exceeds a certain threshold, the method is retranslated with all optimizations and conditional inlining enabled.

3 Path Profiling Technique

In this work, we use a low overhead path profiling technique called bit tracing [5] which is described below. The input to the path profiler is the CFG of a method. The nodes of the CFG are basic blocks. We identify the nodes that are headers of the method or loop headers (i.e., nodes that have an incoming back edge) or exit nodes (nodes that exit out of the method) as *store nodes*. A *store node* stores an encoding of the path starting from the previous store node to the current store node. The store node then initializes the path string by inserting a '1' into it. Code to left shift bits into the path string based on the branch outcome are introduced along the branch edges as depicted in Figure 1.

In Figure 1, block A is the store node since it has an incoming back edge (loop). Node A initializes the path string to '1'. So now path *ABEIJ* has the encoding '101' since the branch outcomes at *A* and *B* cause the edges *AB* and *BE* to left shift a '0' and '1' respectively, in order.

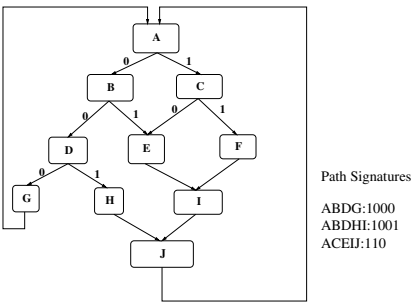

Fig. 1. Path Profiling strategy

The profiling technique employs two registers: one to hold the path encoding so far (referred to as shift register) and the other to hold the address where the path encoding has to be stored. We have used two SPARC registers `g6` and `g7` for these purposes and these registers are not used by LaTTe for register allocation or any other purposes. However since these registers are global we need to store and restore them during each method invocation. This profiling strategy has minimal overhead, as the instrumentation can be added to the CFG in one pass and the profiling information gathered needs minimal processing for hot path prediction. Lastly, this profiling method gives more accurate information about paths than edge profiling methods [5].

Associated with each store node is a circular queue of size n to store the path encodings of the last n recently executed paths. The choice of n depends on the amount of history needed for prediction. In this context, we shall refer to this strategy as n-MRET (n-Most Recently Executed Tails strategy). Apart from the circular queue, a counter is associated with each store node which is incremented each time execution reaches it. If the counter exceeds a particular threshold, referred to as *hotness threshold*, then it implies that the paths originating from the store node have become hot enough to be optimized. A point to be noted is that the hotness threshold may differ from n in that the threshold determines when to predict the hot path, while n determines the amount of profile history seen for making the prediction. A store node stores the previous path encoding

reaching it and then increments its counter. It then loads the store register with the address where the encoding of the next path originating from it is to be stored subsequently.

4 Recompilation Strategy

Our recompilation strategy is as follows:

1. During the first translation, when a method is invoked for the first time, the method is compiled with no optimization.
2. Select hot methods for retranslation using method run counts. Compile these hot methods with all optimizations including method inlining. Instrument the method for gathering path profiles. This retranslation will be referred to as the *first retranslation*.
3. Another retranslation is invoked once enough profile information is available for hot path prediction. Use predicted hot path information for optimization of method along the hot paths. Perform instruction scheduling along the hot paths. Remove profile code introduced in the first retranslation. This retranslation will be referred to as the *second retranslation*.

It should be noted here that in each retranslation, the translation is always from the original bytecode to the native code. To retranslate from a previously translated native code would require retaining many intermediate structures which is considered to be expensive.

The first retranslation is used for two purposes: (i) filtering out methods, called as *cold methods*, that are executed only a few times[1] and (ii) allowing for paths through the method to mature. In our implementation, the first retranslation of a method is invoked whenever the method is invoked more than certain retranslation threshold. The retranslation threshold is a product of hotness threshold and a retranslation factor (referred to as first retranslation factor). In the first retranslation, our instrumentation for performing path profiling is done in the CODE_GEN phase (refer to Section 2), before the SPARC code is generated.

The second retranslation is triggered based on a combination of both method run count and hot path information stored in the store node counter. More specifically, the second retranslation of a method is enabled when either the method run count or the profile count of a hot path in the method exceeds a threshold value. The second retranslation threshold value is the product of hotness threshold and a second retranslation factor. During the second retranslation, the template of the newly generated CFG must match the CFG that was used for instrumentation in the first retranslation. If there is a CFG template mismatch, then the profiling information gathered would be rendered useless. Such a situation could arise in the presence of virtual method inlining. We omit the details here due to space constraints. The reader is referred to [18] for details.

[1] However, there may be many cold methods.

5 Path-Based Optimization

5.1 Region Expansion

A tree region is a unit of local optimizations in LaTTe. Tree regions have a single entry and multiple exits. For example, in Figure 2(a), basic blocks *A, B, C* and *E* fall in a single region. Similarly *D* and *F* are the other two regions in the CFG. In LaTTe, optimizations such as redundancy elimination, common subexpression elimination, constant propagation, loop invariant code motion, as well as local register allocation are performed for tree regions. Hence, bigger the tree regions, greater is the scope of optimization. So, we perform region expansion based on the predicted hot paths.

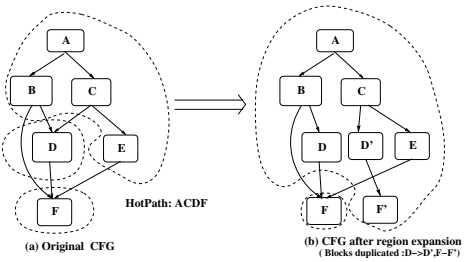

Fig. 2. Region Expansion

In our approach, using the path profile gathered, we duplicate basic blocks that occur in the hot path, to ensure that there are no incoming edges into the hot path. Thus, the region gets expanded and the scope for optimizations is increased. We illustrate this with an example. Consider the CFG in Figure 2(a). Let us assume that path profiling of this CFG yields the information that *ACDF* is the most frequently executed path through it. Since LaTTe does region based optimizations and basic blocks *D* and *F* fall in different regions from nodes *A* and *C*, the path is not optimized completely with respect to the optimizations performed.

In our approach, we clone the blocks *D* and *F* to create two duplicate basic blocks D' and F'. Now in the modified CFG shown in Figure 2(b), the region with header as node A contains the nodes A, B, C, D, E, D' and F'. Since now the hot path lies entirely within the region, the path will be optimized completely with respect to the optimizations performed. In order to avoid excessive code duplication, we use a threshold called *cloning threshold*. Only if the store node associated with the hot path has executed more number of times than the cloning threshold, cloning is initiated.

5.2 Superblock Scheduling

The steps involved in superblock scheduling are superblock formation and list scheduling [10]. It is in superblock formation that the profile information is put to use. The basic blocks that are along a hot path are used to form a superblock. The branch targets, which are predicted to be *taken* based on profile information, are placed on the fall-through path of the branch. Then, the superblock is said to be the sequence of instructions from one join point (a point where control enters through more than one edge) up to the next join point. Superblock formation

has an added advantage in that the predicted branch targets (along the hot path) are not likely to miss on the instruction cache, thereby improving cache performance. List scheduling [16] has been chosen as the scheduling mechanism because it is efficient and at the same time, has a low time overhead. Instructions are scheduled cycle by cycle, based on dependences, till all the instructions are scheduled.

Certain features of our implementation of the superblock scheduler in LaTTe are:

1. It is performed in the OPT phase of the translation (refer to Section 2) after register allocation.
2. It schedules instructions for a superscalar processor.
3. In the current implementation, instructions are not moved above a branch, since this requires live register analysis, which is expensive in terms of runtime overhead.

6 Experimental Results

This section describes our experimental setup and reports the performance of our profile-aided recompilation method. We consider the performance of our technique both with and without superblock scheduling. We compare these with the performance of the original LaTTe JVM (without our profile aided retranslation mechanism.)

6.1 Experimental Setup

We have used the open source LaTTe JIT compiler version 0.9.0 for our implementation [19]. Our test machine is a Sun Ultra 1 workstation having a UltraSPARC I 167 MHz processor with 32 KB L1-cache (16 KB I-cache and 16 KB D-cache), 512 KB L2-cache and 64 MB RAM and running SunOS 5.7. We used 5 benchmarks from SPEC JVM98.

We set the threshold values in our implementation as follows. The hotness threshold, used in the first and second retranslations, of a method is inversely proportional to the number of branches (bytecode branches) in the method. Setting the hotness threshold this way facilitates early retranslation for methods that have larger number of branches or paths. This threshold value is kept the same for our profile-aided retranslation (for both first and second retranslations) and for the original adaptive LaTTe. The first and second retranslation factors (discussed in Section 4) range from 0.1 to 1.0. By tuning these factors, we can control the thresholds for methods to become hot and for paths within the methods to become hot enough for prediction.

6.2 Experimental Results

First we present the reduction in the number of regions along the hot path when profile-aided retranslation and region expansion were performed. Table 1 reports

the number of regions across which the basic blocks in the hot paths were spread over before and after region expansion. In this experiment, we have chosen the multiplication factors for method run count threshold in the first and second retranslation to be 0.1 and 0.5 respectively. We used a hotness threshold of 20 and cloning threshold of 15. After region expansion, a hot path lies entirely within a single region. The table indicates that our region expansion technique significantly decreases the number of regions in which the hot paths lie by a factor four or five times. Thus region expansion exposes more scope for aggressive optimizations on hot regions.

Next we present the execution times of SPEC JVM programs in LaTTe with and without profile aided recompilation. We refer to LaTTe with profile aided recompilation as P-LaTTe. In this experiment, we have chosen 0.2 as the multiplication factor for the first retranslation and 1.0 as the second retranslation factor in P-LaTTe, as against adaptive LaTTe which uses the multiplication factor 0.2 in its first retranslation. Further, in P-LaTTe, the hotness threshold for the store nodes is chosen to be 20 and the cloning threshold is set to be 15. With these parameters, more time is given for profiling, and the second retranslation is triggered only after "enough" profile is gathered.

The results of this experiment are mixed. We find that the execution times under LaTTe and P-LaTTe (with and without instruction scheduling) are comparable (refer to Table 2), in spite of the retranslation, profiling, and cloning overheads. First we note that the comparison between original LaTTe with and without adaptive retranslation also gives mixed results. More specifically, in 2 out of the 5 benchmarks, original LaTTe with adaptive retranslation performs poorly compared to original LaTTe without adaptive retranslation. Our P-LaTTe with instruction scheduling performs better only in one benchmark, namely `_222_mpegaudio`, and relatively poorly in the other four. Comparing the execution times of the programs under P-LaTTe with and without instruction scheduling, we observe that the benefits due to instruction scheduling compensates fairly well, though they are not significantly high to result in better performance compared to the original LaTTe. One reason for the improvement in performance due to instruction scheduling being small is the fact that our instruction scheduler is a post-pass scheduler, i.e., it performs scheduling after

Table 1. Number of regions containing the hot paths before and after region expansion

Benchmark	Before region expansion	After Region expansion
_202_jess	466	104
_201_compress	165	44
_209_db	181	44
_222_mpegaudio	192	31
_228_jack	506	105

Table 2. Performance comparison with execution times: Original LaTTe, Original Adaptive LaTTe, P-LaTTe, and P-LaTTe with Instruction Scheduling

Benchmark	Original LaTTe	Original Adaptive LaTTe	P-LaTTe	P-LaTTe with instruction scheduling
_201_compress	88.07 s	85.94 s	90.30 s	90.01s
_202_jess	53.71 s	52.22 s	54.97 s	55.48s
_209_db	85.56 s	91.59 s	92.75 s	91.99s
_222_mpegaudio	76.60 s	75.50 s	77.06 s	75.04s
_228_jack	58.82 s	63.11 s	66.05 s	66.40s

register allocation. As in any postpass scheduling, register allocation introduces anti- and output-dependences which somewhat restricts the parallelism exposed by the instruction scheduler. Thus, we conclude that our results are encouraging in the sense that the performed optimization results in execution times which are comparable to that of the original LaTTe, despite the additional profiling and retranslation overhead. To obtain greater benefits, additional profile-based optimizations could be tried.

In order to estimate the overheads involved in our method, we measured the time taken for the second retranslation. Table 3 presents the total retranslation overheads with and without instruction scheduling. The (second) retranslation overhead without instruction scheduling is significantly higher in two of the benchmarks, viz., `_202_jess` and `_228_jack`. Note that, in these benchmarks, P-LaTTe performed relatively worse than adaptive LaTTe. The considerable retranslation overhead comes from the fact that both first and second retranslations still translate code from bytecode to native code, rather than from native code to native code. Due to this, the performance gain achieved from our optimizations is lost. We also observe that the additional overhead for performing superblock scheduling is very low.

Table 3. Retranslation time for profiling only, profiling with instruction scheduling

Benchmark	Profiling only	Profiling with instruction scheduling
_201_compress	0.30 s	0.37 s
_202_jess	2.21 s	2.59 s
_209_db	0.47 s	0.58 s
_222_mpegaudio	0.54 s	0.78 s
_228_jack	3.02 s	3.80 s

7 Related Work

Several recent research projects have focused on aspects of dynamic optimization [4, 8, 9, 12]. Dynamo [4], a dynamic optimization software, attempts to run a statically optimized binary faster. This system uses the Most Recently Executed Tail (MRET) strategy [11], in which the path that is executed after

a start of trace node has become hot is predicted as the hot path. Our n-MRET strategy reduces the probability of noise paths (infrequently executed paths) being selected as the hot path which is more probable in MRET. Similar work on profile-driven dynamic recompilation for Scheme makes use of Ball and Larus edge profiling algorithm [8]. We have chosen path profiling over edge profiling because of more accurate information available in the former. The theoretic and algorithmic results, which may be used to determine when an edge profile is a good predictor for hot paths and when it is a poor predictor, are discussed in [7]. The efficient path profiling algorithm by Ball and Larus has a high instrumentation overhead though it has the least profiling overhead of all path profiling algorithms.

For choosing hot methods, LaTTe relies on method run counts. Alternatively, time based and sample based profiling can also be used [3]. The IBM Jalapeno optimizing compiler [9] for Java uses sample based profiling for determining hot methods for retranslations. In this case, the program counter (PC) is sampled at regular intervals and this gives a more accurate measure than mere run counts, since long-running methods with less number of invocations are also accounted for in this strategy.

8 Conclusions and Future Work

In this work we have implemented a dynamic path profiling strategy for aiding recompilation in the LaTTe JIT compilation framework. Our profile aided recompilation mechanism and region expansion reduces the number of regions significantly, by a factor of 4 to 5, which increases the scope for performing aggressive compiler optimization on hot methods and hot paths. With profile assisted compilation and instruction scheduling, our implementation performs comparable to the original adaptive LaTTe in spite of the retranslation and profiling overheads. Future work could concentrate on avoiding a complete retranslation by retaining certain structures from the previous translation of the method. Another future work could be on developing heuristics that vary the cloning threshold for different paths, instead of a fixed (static) threshold value. Last, using the profile information to perform other optimizations along the hot paths could be explored.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast effective code generation in a Just-In-Time Java compiler. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, June 1998. 496

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, P. F.Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, Oct. 2000. 496

[3] M. Arnold, M. Hind and B. G. Ryder. An empirical study of selective optimization. In *Proc. of the 13th Intl. Workshop on Languages and Compilers for Parallel Computing*, 2000. 504

[4] V. Bala, E. Duesterwald, S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the SIGPLAN '98 Conf. on Programming Language Design and Implementation*, 2000. 503

[5] T. Ball and J. R.Larus. Optimally profiling and tracing programs. In *Proc. of the 19th Symp. on Principles of Programming Languages*, Jan. 1992. 496, 498

[6] T. Ball and J. R.Larus. Efficient path profiling. In *Proc. of 29th Symp. on Microarchitecture*, Dec. 1996. 496

[7] T. Ball, P. Mataga and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Jan. 1998. 496, 504

[8] R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. of Intl. Conf. on Computer Languages (ICCL'98)*, May 1998. 503, 504

[9] M. G.Burke, D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proc. of 1999 ACM Java Grande Conference*, June 1999. 503, 504

[10] W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, W. W. Hwu. Profile-assisted instruction scheduling. *International Journal of Parallel Programming*, 1994. 496, 500

[11] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proc. of the 9th Intl. Conf. on Architecture Support for Programming Languages and Operating Systems*, 2000. 496, 503

[12] B. Grant, M. Mock, M. Philipose, C. Chambers and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report UW-CSE-97-03-03, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, 1997. 503

[13] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective structure for VLIW and superscalar compilation. *Journal of Supercomputing*, Feb. 1993.

[14] *Java HotSpot Performance Engine*, http://java.sun.com/products/hotspot/, 1999. 496

[15] J. Lee, B-S. Yang, S. Kim, S. Lee, Y. C. Chung, H. Lee, J. H. Lee, S-M. Moon, K. Ebicioglu, E. Altman. Reducing virtual call overheads in a Java VM Just-in-Time compiler. In *Proc. of 1999 Workshop on Interaction between Compilers and Computer Architectures*, Jan. 2000. 497

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997. 501

[17] *The SPARC Architecture Manual Version 8*. 497

[18] R. Vinodh Kumar. Dynamic path profile aided recompilation in a Java Just-In-Time compiler M. E. Dissertation Project, Indian Institute of Science, Dept. of Computer Science & Automation, Bangalore, 560 012, India, Jan. 2000. 499

[19] B-S. Yang, S-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung,S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-In-Time compiler with fast and efficient register allocation. In *Proceedings of the 1999 Intl. Conf. on Parallel Architectures and Compilation Techniques, Oct. 1999*. 496, 497, 501

[20] F.Yellin and T.Lindholm, *The Java Virtual Machine Specification*, Addison-Wesley, 1996. Hall, 1999. 495