Evaluating the Impact of Thread Escape Analysis on a Memory Consistency Model-Aware Compiler

Chi-Leung Wong[1], Zehra Sura[2], Xing Fang[3], Kyungwoo Lee[3], Samuel P. Midkiff[3], Jaejin Lee[4], and David Padua[5]

[1] KAI Software Lab, Intel Americas, Inc., Champaign, IL, USA
chi.leung.david.wong@intel.com
[2] IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA
zsura@us.ibm.com
[3] Purdue University, West Lafayette, IN, USA
{xfang,kwlee,smidkiff}@ecn.purdue.edu
[4] Seoul National University, Seoul, Korea
jlee@cse.snu.ac.kr
[5] Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
padua@cs.uiuc.edu

Abstract. The widespread popularity of languages allowing explicitly parallel, multi-threaded programming, e.g. Java and C#, have focused attention on the issue of *memory model* design. The Pensieve Project is building a compiler that will enable both language designers to prototype different memory models, and optimizing compilers to adapt to different memory models. Among the key analyses required to implement this system are *thread escape analysis*, i.e. detecting when a referenced object is accessible by more than one thread, *delay set analysis*, and *synchronization analysis*.

In this paper, we evaluate the impact of different escape analysis algorithms on the effectiveness of the Pensieve system when both delay set analysis and synchronization analysis are used. Since both analyses make use of results of escape analyses, their precison and cost is dependent on the precision of the escape analysis algorithm. It is the goal of this paper to provide a quantitative evalution of this impact.

1 Introduction

In shared memory parallel programs, different threads of the program communicate with each other by reading from and writing to shared memory locations. Experience shows that to achieve high performance without extensive analyses, it is necessary to allow memory accesses to follow an order of execution that is non-intuitive one[13]. Memory system behavior observed by different processors constitute the memory model. It is difficult to define a memory model that is both easy to use and implement efficiently. The goal of the Pensieve compiler system is to provide a testbed to evaluate memory models by creating "virtual" memory models and to evaluate the overhead of these models in the presence of aggressive compiler analyses and optimizations. Given a program and a memory model specification, the Pensieve compiler will ultimately be able to generate different versions of machine code corresponding to the specified memory model. However, the current version of the Pensieve system only creates a sequentially consistent "virtual" memory model and implements it on the Intel IA32 and PowerPC processors, so the virtual memory model and the target memory models are currently hardwired inside the system. An important issue in the system design is performance — both the compilation time and application time should be minimized. In this paper, we investigate the impact of escape analysis on our Pensieve system. We study how escape analysis affects the cost and precision of other analysis algorithms, which in turn affects both the compilation cost and application performance. In particular, this paper makes the following contributions:

– it describes the Pensieve compilation system;
– it describes the interaction between escape analysis and synchronization/ delay set analyses.
– it presents a quantitative study on the impact of escape analysis on the Pensieve system.

1.1 Memory Models

A memory model[1] specifies the memory system behavior, and can be specified for programming languages as well as hardware. Memory models are necessary because they define the allowable set of outcomes of a parallel program and, as a result, they allow programmers to reason about their programs and compilers to generate valid code. Until recently, memory models were of concern only to expert systems programmers, and computer architects. With the advent of languages like Java and C#, many programmers write multi-threaded programs targeting Internet, database, and GUI applications, in addition to traditional high performance computing applications. Because of this, memory models have become an issue for much of the programmer community and for language and compiler designers. The trade-offs between ease-of-use and performance have become increasingly important.

Sequential Consistency. A well-known memory model is sequential consistency (SC), defined by Lamport[15]. It is often considered to be the simplest and most intuitive memory consistency model [13]. Scheurich and Dubois[19] described a sufficient condition for SC and Gharachorloo et. al.[8] presented the condition in a slightly difference way. The idea of these sufficient conditions is to delay a memory access until all previous ones within the same thread are completed. These conditions impose constraints so that some performance improving optimizations cannot be applied in the hardware . In addition, it constrains

[1] Memory models are often called consistency models in the context of hardware.

compiler optimizations that may reorder memory accesses. The issue of memory models can be illustrated by the busy-wait synchronization example shown in Figure 1(a). Both `x` and `a` are shared variables accessible by two concurrent threads. Thread 1 does some computation and stores the result in `a`, and then uses `x` to inform Thread 2 that a *new* value of `a` is ready to be read. Thread 2 waits for the data by executing a `while` loop that reads `x` and waits for the value to become non-zero, at which time the thread will read the value from `a`. The program shown in Figure 1(a), if executed in a SC environment, achieves the described intention.

Relaxed Consistency Models. Most multiprocessor systems implement consistency models, such as weak ordering and release consistency [4], which impose fewer constraints than SC on the order of shared memory accesses. Where clear, we will refer to these more relaxed models by the acronym RC. RC models allow more instruction reordering, increasing the potential for instruction level parallelism and as a result can potentially deliver better performance. Synchronization primitives, such as *fences*, are used in these systems to force an order on memory operations that is more constrained than that implied by the default consistency model.

The program shown in Figure 1(a), if executed in a RC environment, is not guaranteed to achieve the programmer's intention. This is because, for performance reasons, the compiler or hardware may reorder the two memory operations performed by Thread 1 such that the update of `x` reaches Thread 2 *before* the update of `a`. If this happens, `T2` could read the updated value of `x` (i.e. 1), exit the loop, and then read an *old* value (i.e. 0) of `a`. Therefore, the intention of the programmer is not achieved. In the presence of the `fence` instruction, the memory reording does not happen. Figure 1(b) shows a correct implementation of the busy-wait construct using fences.

Both `x` and `a` are zero initially.

```
// Thread 1   // Thread 2

     .             .
     .             .
     .             .
S1: a = 1;    T1: while (x==0) wait;
S2: x = 1;    T2: print a;
```

(a) Busy-wait synchronization

```
     .             .
     .             .
     .             .
U1: a = 1;    V1: while (x==0) wait;
U2: fence     V2: fence
U3: x = 1;    V3: print a;
```

(b) Fence instruction inserted to avoid reordering

Fig. 1. Memory model issues example

1.2 Enforcing Memory Models

Enforcing a memory model implies enforcing some memory access orders. However, not all orderings specified by the memory model need to be enforced. In fact, only those orderings that may affect the outcome of the program must be enforced. To generate efficient and correct code, a compiler must determine which memory accesses may not be reordered and enforce only those orderings. The orderings that must be enforced are called *delays*. In [20], Shasha and Snir give minimal criteria for which orders must be enforced in order to have a sequential consistent execution of a program. Both [20] and this paper assume that the hardware provides primitives, such as fences, powerful enough to enforce the required orderings. Moreover, some compiler optimizations must be constrained if applying them may violate a delay. In [20], the authors present a delay set analysis (DSA) algorithm to determine the required orderings. DSA requires the thread structure of the programs to determine the delay information.

In Section 2, we describe the Pensieve system design. In Section 3, we describe the escape analysis proposed in [23]. In Section 4 and Section 5, we describe how the escape analysis impact delay set analysis and synchronization analyses respectively. In Section 6, experimental results are presented to evaluate the impact of escape analysis quantitatively. This paper concludes in Section 7.

2 Pensieve Compiler System Design

Our Pensieve Compiler System supports SC on top of two hardware platforms that support more relaxed memory models — the Intel platform and the PowerPC platform, which is an extension of the Jikes RVM infrastructure [7,9]. Figure 2 gives an overview of the Pensieve system.It shows three phases:

1. In the analysis phase, a set of delays is computed. The delays are the ordering constraints to be enforced both by the compiler and the hardware.
2. In the modified code optimization phase, the set of delays identified by the analysis phase is checked before performing an optimization transformation. If a transformation would violate a delay, it is not applied.
3. In the fence insertion and optimization phase, fences are inserted into the program to force the delays to be enforced by the hardware. This phase looks for opportunities to synchronize multiple delays with a single fence instruction. The details of this phase are described in [10,11].

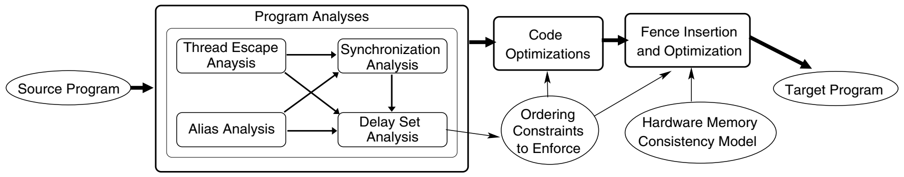


Fig. 2. Overview of the Pensieve system

3 Thread Escape Analysis

Thread escape analysis aims at identifying objects which *may* be accessed by two or more threads. In the Pensieve System environment, the analysis is performed as the application programs are running, so the time to perform escape analysis is a part of the overall execution time. Therefore, an inexpensive and moderately accurate analysis algorithm will be a good choice in our approach. In this project, we balance analysis algorithm performance and accuracy. While we are not aiming at having an escape analysis that is precise for the whole program, the analysis should be precise enough that fences are not unnecessarily inserted into frequently executed methods. In light of this, we chose to design the simplest possible algorithm to minimize the cost of the analysis. In the Pensieve compiler system, we have implemented four escape analysis algorithms:

- a *connectivity* based analysis described in [23]
- a *field* based analysis described in [22]
- Bogda's analysis described in [6]
- Ruf's analysis described in [18]

3.1 Connectivity Based Analysis

The basic characteristics of the algorithm[23] are:

- Analysis of most memory accesses is field insensitive, with accesses in `Runnable` objects being field sensitive.
- More precise context information is constructed for the `run()` method of a `Runnable` class (i.e. `this` is not assumed to escape) than for other methods.
- Objects assumed to be reachable by multiple threads, are marked as escaping only if they are *accessed* by multiple threads.

The analysis is a two-phase analysis. The **bottom-up phase** computes the effect of methods and computes how the methods make arguments escaping. The **top-down phase** computes the context of methods and determines how the caller makes arguments escaping before passing them to their callees. Both phases are done by visiting the strongly connected component (SCC) graph induced by the call graph in (reverse) topological order. The analysis makes use of the union-find data structure to avoid fixed point computations for recursive methods within an SCC.

3.2 Field Based Analysis

The basic characteristics of the algorithm[22] are:

- Analysis of all objects is field sensitive. To avoid an expensive analysis, unlike [18], it merges escaping properties of fields of all objects of the same type. For example, if O_1`.f` $= O_2$ and O_2 is found to be escaping, then for any object O, if O is referened by a field `f`, it is assumed to be escaping.
- Analysis of the `run()` method of a `Runnable`, looks for conditions implying `this` is not escaping, instead of assuming `this` is escaping. .

The analysis is an iterative analysis — the analysis is performed until no escaping properties of variables and fields change. It is a partially context sensitive analysis.

3.3 Bogda's Analysis

Bogda's analysis[6] is a two phase and iterative analysis. The basic characteristics of the algorithm are:

- an object is escaping if any of the following conditions is fulfilled
  - it is reachable via more than one field reference;
  - it is reachable by a static field; or
  - it is reachable by a `Runnable` object.

3.4 Ruf's Analysis

Ruf's analysis[18] is a three phase analysis. Like our connectivity based analysis, it makes use of the union-find data structure to avoid fixed point computations for recursive methods inside an SCC. The basic characteristics of the algorithm are:

- an object is escaping if it is both
  - reachable from static fields or `Runnable` objects;
  - synchronized by more than one thread.

Since the analysis is designed for synchronization removal, we have adapted it for fence insertion. Instead of using the second condition "synchronized by more than one thread", the adapted analysis checks whether an object is "accessed by more than one thread". After the adaptation, the cost of analysis could be increased because there are more object accesses than synchronization operations.

4 Impact of Escape Analysis on Delay Set Analysis

Delay set analysis computes a *delay set*, i.e. a set of ordered pairs of memory access (x, y) such that y must be delayed until x has completed. In [20], Shasha and Snir present an accurate method to find the minimal delay set. In the Pensieve compiler system, we use a much simpler approximate method described in[21]. The analysis in [20] finds cycles in a graph where nodes are shared variable accesses from two or more threads. In our simplified escape analysis, we look for pairs of shared memory accesses (x, y) such that x precedes y; y is aliased to y' in another thread; x is aliased to x' in another thread; and y' precedes x'.

Escape analysis affects both the precision and cost of delay set analysis. The fewer the number of escaping variables, the fewer pairs (x, y) that need to be checked, and the fewer the number of x' and y' accesses. This increases both the speed and the precison of delay set analysis.

5 Impact of Escape Analysis on Synchronization Analysis

Synchronization information helps reduce the number of conflict edges in the graph considered for delay set analysis, and thus improves the precision of delay set analysis[14].

In our analysis, we consider the following Java synchronization primitives:

- `synchronized` blocks, used for lock-based synchronization
- thread `start()` and `join()` calls, used to determine the program thread structure.

Our lock-based synchronization analysis has been described in [22]. It improves the accuracy of our approximate delay set analysis. In essence, we can ignore pairs of nodes (x, y) and (x', y'), as described above, when both are synchronized with the same lock. See [22] for details.

A detailed description of our start-join-based synchronization analysis is given in [21]. The idea is to make use of the Java language semantics of `start()` and `join()`. When a thread is spawned via a thread `start()`, all memory accesses of the creator thread that are initiated before `start()`, complete before the point where the new thead starts. Also, if a thread T invokes a `join()` call to wait for another thread to terminate, then all memory accesses performed by the terminating thread complete before T continues execution after the `join()`.

Escape analysis affects the precision of synchronization analysis. When doing synchronization analysis, we consider only `join()` calls that are matched with some `start()` call. A `join()` is matched with a `start()` only if the objects that they are invoked on do not escape. Matched `join()` calls can reduce the number of pairs (x, y) to be considered. Therefore, when escape information is more precise, more `join()` calls can be matched, so more pairs (x, y) can be ignored.

6 Experimental Results

In this section we present the results of executing benchmark programs compiled with our Pensieve compiler using the four escape analyses described in Section 3. Our goal is to quantitatively evaluate the impact of different escape analysis algorithms.

6.1 Benchmark Programs

Table 1 shows the benchmark programs used in the experiments. These are standard benchmarks from the SPECjvm98, SPECjbb2000 and the Java Grande benchmark suite. There are also some programs taken from the literature, including the concurrent implementation of two data structures, hashmaps and

Table 1. Benchmark Characteristics

Benchmark	Description	Source	# bytecodes
moldyn	Molecular dynamics application	Java Grande Forum Multithreaded Benchmarks[3]	26,913
montecarlo	MonteCarlo simulation	Java Grande Forum Multithreaded Benchmarks[3]	63,452
raytracer	Ray tracing application	Java Grande Forum Multithreaded Benchmarks[3]	33,198
mtrt	Ray tracing application	From the SPECjvm98 benchmark suite[2]	290,260
boundedbuf	Producer-consumer application	Uses Doug Lea's Blocking Queue class[16]	12,050
geneticalgo	Parallel genetic algorithm	Adapted from the sequential version version in [16]	30,147
hashmap	Microbenchmark for concurrent hashmaps	Uses Doug Lea's ConcurrenthashMap class[16]	24,989
seive	Sieve of Erastothenes	From an example in [12]	10,811
disksched	Disk scheduler using an elevator algorithm	From an example in [17]	21,186
jbb	Middle-layer database server application	SPECjbb2000[1]	521,021

queues. These concurrent data structures are expected to be widely used and have been incorporated in the Java standard libraries.

6.2 Target Architectures

The experiments are performed on two platforms — the Intel IA32 platform and the PowerPC platform:

- The Intel platform is a Dell PowerEdge 6600 SMP with 4 Intel 1.5Ghz Xeon processors with 1MB cache each, and 6G system memory.
- The PowerPC platform is an IBM SP 9076-550 with 8 375Mhz processors with 8GB system memory.

6.3 Software Settings

Our compiler system is implemented on top of the Jikes Research Virtual Machine [7,5,9] version 2.3.4. We use the FastAdaptiveSemiSpace configuration with no fences inserted within the virtual machine code. For the experiments reported below, we force the system to use the optimizing compiler. To evaluate the impact of escape analyses, we compare the analysis times of delay set analysis and synchronization analysis. In addition, we compare the precision of delay set analysis and synchronization analysis w.r.t. different escape analyses by comparing the application execution time and the number of fences inserted. In all the graphical plots, the geometrical means are included to summarize data for all the benchmark programs.

There are six escape analyses compared:

- `empty` assumes all memory accesses are escaping accesses.
- `argEscape` assumes all memory locations reachable from some arguments are escaping.
- `connect` is the connectivity based escape analysis algorithm described in Section 3.
- `field-based` is the field based escape analysis algorithm described in Section 3.
- `bogda` is Bogda's escape analysis algorithm described in Section 3.
- `ruf5` is Ruf's escape analysis algorithm described in Section 3.

6.4 Cost of Escape Analysis

Figure 3 presents the time taken using a log scale for performing escape analysis. The times for `empty` and `argEscape` are small because they are very simple. Other than these two trivial analyses, the connectivity based analysis is the fastest because it does not require a fixed-point computation. It takes longer than `empty` and `argEscape` because it is an interprocedural analysis. The analysis times of `field-based` and `bogda` are longer because they are interprocedural iterative analyses that requires a fixed-point computation. On average, the analysis time of `ruf5` is between those of `connect` and `field-based`.

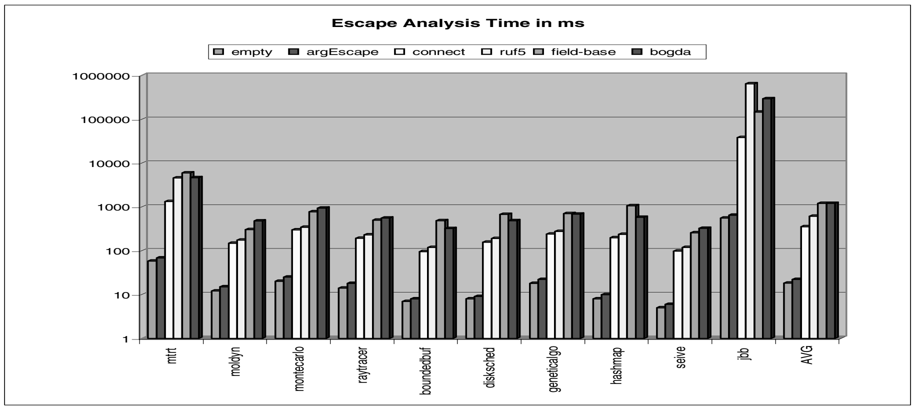


Fig. 3. Escape analysis time in msec

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	62371	54240	200307	207529	243376	247371
moldyn	11782	297740	297782	298239	308673	309121
montecarlo	22132	3583	4101	7095	31766	31847
raytracer	17768	46960	48967	49116	63153	63539
boundedbuf	2599	4498	4733	4778	6163	6163
disksched	4855	5394	5425	5791	7748	7748
geneticalgo	9574	17126	18282	16877	26952	26952
hashmap	4030	4134	4274	4274	4972	4972
seive	2668	4925	4925	5139	5525	5525
jbb	1872250	916591	832800	836559	1847126	1852503

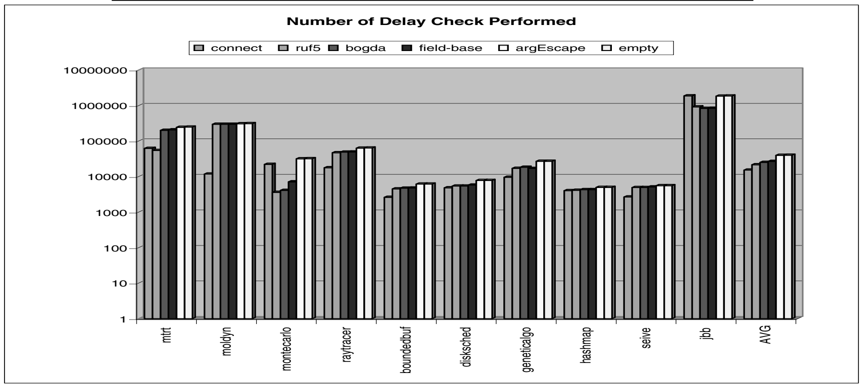


Fig. 4. Number of delay checks

6.5 Impact on the Cost of Delay Set Analysis and Synchronization Analysis

We evaluate the impact of escape analysis on delay set analysis and synchronization analysis separately. In both cases, we measure the time taken to perform these two analyses. In case of delay set analysis, we also measure the number of memory access pairs checked for delays.

Figure 4 shows the number of delay checks for different escape analysis algorithms. Since the value range is huge, it is plotted using a log scale. We can see `connect` analysis lead to fewer checks than other escape analyses for most benchmarks except `mtrt`, `montecarlo` and `jbb`. By comparing `connect` and `field-based` for benchmarks `montecarlo` and `jbb`, we can see that in these benchmarks, being field sensitive is important. On average, `connect` leads to

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	58.41	56.25	93.52	90.38	104.20	108.35
moldyn	1.90	46.26	47.04	47.26	49.53	49.61
montecarlo	9.53	1.77	1.88	2.27	11.13	10.48
raytracer	2.80	8.14	8.53	8.70	11.05	11.07
boundedbuf	0.42	0.70	0.71	0.69	0.89	1.83
disksched	0.85	1.03	1.03	1.09	1.46	1.46
geneticalgo	1.93	3.59	3.68	3.41	5.25	4.76
hashmap	0.59	0.69	0.72	0.71	0.79	0.83
seive	0.87	1.16	1.19	1.19	1.48	1.30
jbb	304.09	144.76	127.21	127.43	297.69	295.74

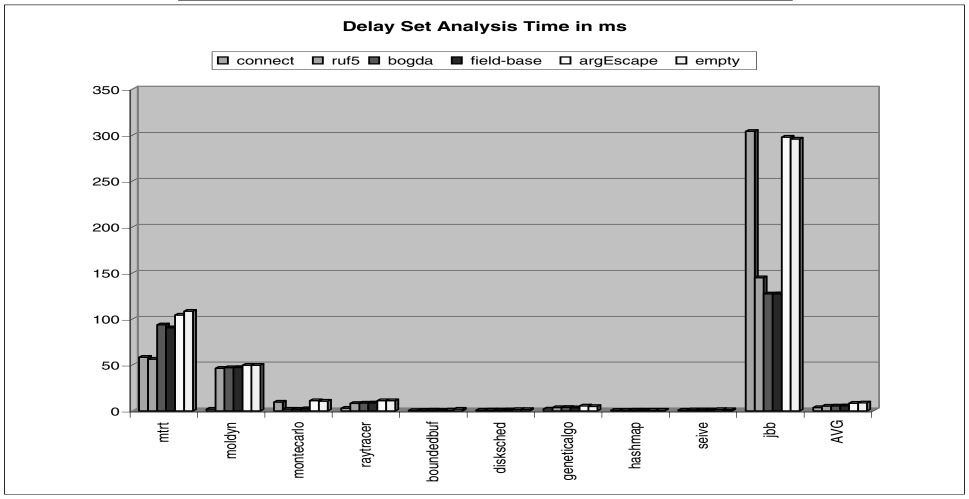


Fig. 5. The time spent on delay set analysis in msec

	field-based	bogda	empty	argEscape	connect	ruf5
mtrt	478.30	829.88	873.38	905.41	841.13	839.60
moldyn	73.85	122.74	133.15	132.14	130.63	132.87
montecarlo	270.28	343.19	359.99	362.02	358.69	349.72
raytracer	134.14	188.00	198.61	200.09	200.85	190.60
boundedbuf	67.57	118.79	117.81	117.04	125.85	123.36
disksched	103.17	160.92	180.60	182.97	165.02	161.78
geneticalgo	159.78	228.40	248.58	247.21	248.88	251.44
hashmap	139.50	237.22	251.10	251.19	252.38	247.33
seive	38.18	74.56	76.94	76.75	76.04	75.48
jbb	56070.55	74231.65	59977.42	58466.61	72130.53	133368.70

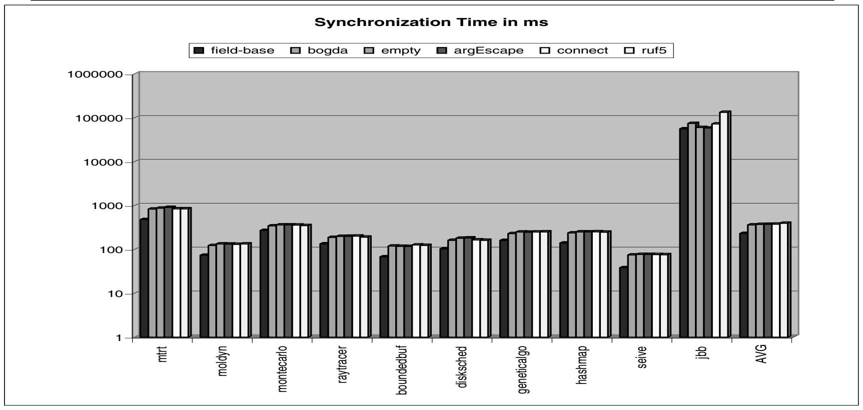


Fig. 6. The time spent on synchronization analysis time in msec

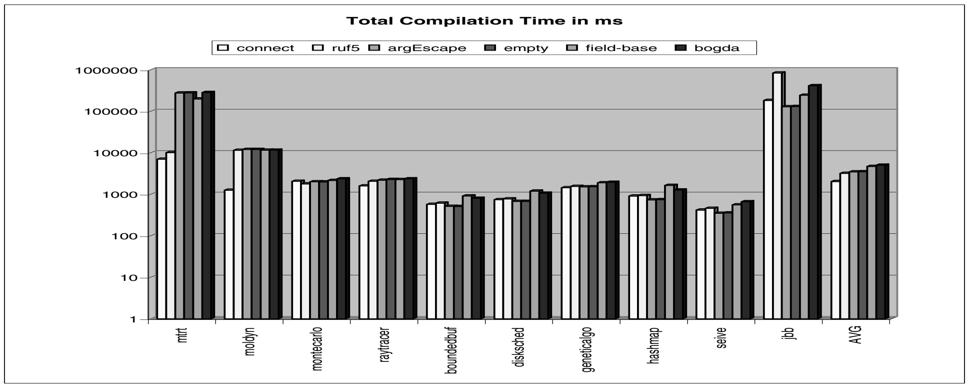


Fig. 7. Total Compilation Time in msec

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	61168	52331	198331	205700	240965	244926
moldyn	10312	294371	294409	294409	302913	302913
montecarlo	10410	1449	1544	2707	18182	18263
raytracer	16477	41063	41767	41767	52902	53288
boundedbuf	1468	2596	2764	2625	3067	3067
disksched	3590	4074	4074	4441	5923	5923
geneticalgo	6802	12846	12846	12780	14771	14771
hashmap	1871	2031	2075	2075	2158	2158
seive	1545	3150	3150	3150	3439	3439
jbb	1050402	252850	265206	264630	962122	965368

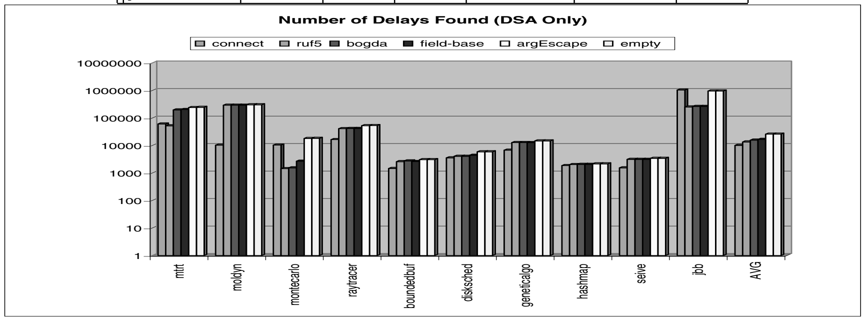


Fig. 8. The number of delays found (delay set analysis only)

fewer checks than other escape analyses. A similar pattern is observed for the delay set analysis times shown in Figure 5.

Figure 6 shows the synchronization analysis time. We can see the analysis times for synchronization analysis are similar for `bogda`, `empty`, `argEscape`, `connect` and `ruf5`. We observe that `field-based` leads to faster synchronization analysis on all benchmarks. In our system implementation, `field-based` shares some data structures with synchronization analysis, so synchronization analysis reuses data computed by `field-based`. We expect these data reuses reduce the synchronization analysis time.

The total compilation time is shown in Figure 7. We observe that, on average, `connect` outperforms other non-trivial escape analysis algorithms in this aspect.

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	61160	52323	197468	204837	239914	243875
moldyn	10312	294324	294362	294362	302866	302866
montecarlo	9846	1406	1501	2664	17618	17699
raytracer	16477	40945	41648	41648	52783	53169
boundedbuf	1468	2596	2764	2625	3067	3067
disksched	3590	4069	4069	4436	5918	5918
geneticalgo	6802	12835	12835	12769	14760	14760
hashmap	1871	2028	2072	2072	2155	2155
seive	1541	3146	3146	3146	3435	3435
jbb	1049689	252753	265108	264532	961432	964669

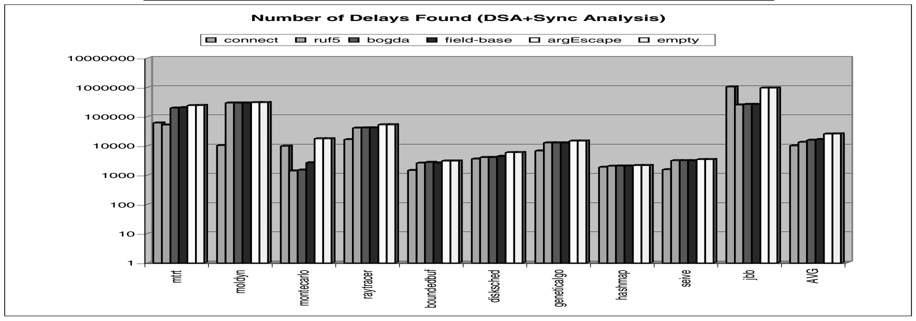
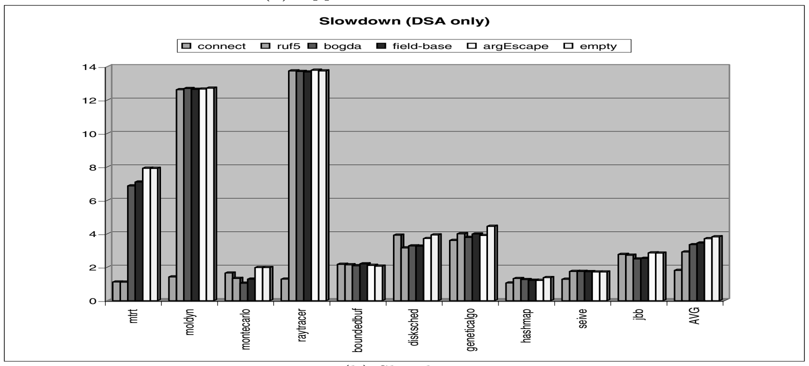


Fig. 9. The number of delays found (delay set analysis + synchronization analysis)

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	3.80	3.78	23.47	24.26	27.08	27.06
moldyn	74.08	659.89	663.48	660.92	661.80	664.59
montecarlo	119.25	96.52	75.79	93.09	143.04	143.33
raytracer	74.94	798.80	796.49	795.08	801.13	798.53
boundedbuf	1484.75	1467.53	1430.16	1506.80	1443.88	1407.05
disksched	5.83	4.70	4.88	4.86	5.54	5.86
geneticalgo	53.22	59.16	55.91	58.92	57.62	65.74
hashmap	42.07	52.63	50.12	48.45	48.20	55.05
seive	160.79	219.55	220.04	217.83	214.93	215.61
jbb	4346.56	4419.63	4822.99	4746.00	4206.01	4231.55

(a) Application execution time



(b) Slowdown

Fig. 10. Slowdown due to fence instruction insertion (delay set analysis only)

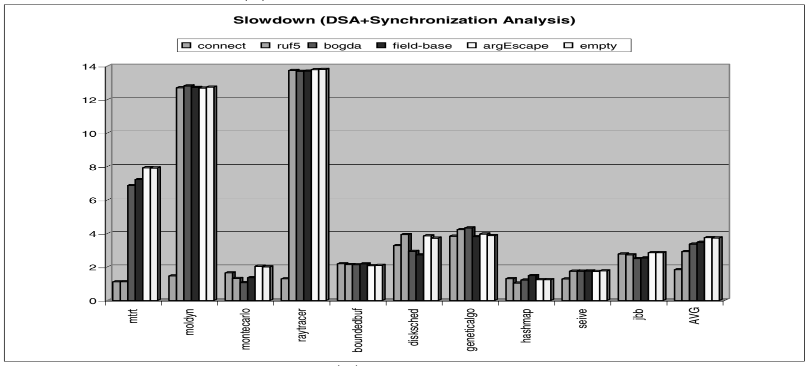
6.6 Impact on Analysis Precision

The analysis precision of delay set analysis and synchronization analysis can be measured in terms of application execution time and number of delays found. In both cases, we can view the precision in the following cases:

- the performance of delay set analysis (without applying synchronization analysis)
- the performance of delay set analysis with refinement of synchronization analysis

	connect	ruf5	bogda	field-based	argEscape	empty
mtrt	3.77	3.80	23.50	24.70	27.08	27.05
moldyn	76.42	664.30	670.10	665.27	663.29	666.33
montecarlo	119.21	95.26	76.88	98.98	147.86	144.56
raytracer	74.91	798.32	795.36	796.14	801.17	802.04
boundedbuf	1496.27	1464.73	1453.98	1491.85	1415.81	1432.90
disksched	4.89	5.87	4.37	4.05	5.75	5.55
geneticalgo	56.76	62.55	63.92	56.26	58.61	57.24
hashmap	51.78	41.29	48.66	59.47	49.31	49.44
seive	161.18	220.07	220.14	220.81	218.31	221.04
jbb	4330.03	4417.78	4803.98	4741.15	4221.02	4215.59

(a) Application execution time



(b) Slowdown

Fig. 11. Slowdown due to fence instruction insertion (delay set analysis + synchronization analysis)

Figure 8 shows the number of delays found when only delay set analysis is applied. We can see that for most benchmarks fewer delays are found when `connect` is applied. Similar to the pattern described in previous section, `connect` does not outperform other escape analyses for benchmarks `mtrt`, `montecarlo` and `jbb`. We can see a similar pattern when both delay set analysis and synchronization analysis are applied, shown in Figure 9.

Finally, the application execution times are reported in Figure 10 (only DSA applied) and Figure 11 (both DSA and synchronization analysis applied). In both settings, we also plot the slowdown graphs in the same figure. We can see the `connect` performs well for most benchmarks except for `montecarlo`, `disksched` and `jbb`. On average, `connect` is the best analysis from slowdown perspective.

7 Conclusions

In this paper, we have presented the Pensieve Compiler System. The system presented in this paper focuses on enforcing SC on the Intel IA32 and PowerPC platforms. We also presented the interactions between our thread escape analyses, synchronization analysis, and delay set analysis implemented in the system. We can see, on average, the connectivity analysis is the best escape analysis algorithms leading to good application performance. From the analysis time perspective, connectivity analysis is much faster than other non-trivial analyses. Ruf's analysis is the second best analysis that lead to good application performance. For some benchmarks, Ruf's analysis outperforms connectivitiy analysis. However, Ruf'a analysis is much slower than connectivity analysis, so we choose to use `connect` as the escape analysis in the Pensieve system.

By comparing with the field based analysis, we can see the importance of being field sensitive for benchmarks like `montecarlo` and `jbb`. The result motivates further works to design a fast and precise escape analysis to be used by delay set analysis and synchronization analysis by enabling field sensitivity for connectivity analysis without increasing the analysis cost significantly.

References

1. SPEC JBB 2000 Benchmark. URL: http://www.specbench.org/jbb2000.
2. SPEC JVM Client98 Suite. URL: http://www.specbench.org/jvm98/jvm98.
3. The Java Grande Forum Multi-threaded Benchmarks. URL: http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.
4. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.
5. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming and Systems, Languages, and Applications (OOPSLA) 2000*, Minneapolis, MN, October 2000.
6. Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM Press, 1999.
7. B. Alpern et. al. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.
8. Kourosh Gharachorloo et. al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of The 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–26, May 1990.
9. Michael G. Burke et. al. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 129–141, Palo Alto, CA, USA, Jun 1999.
10. Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory processing. In *2003 ACM International Conference on Supercomputing*, June 2003.
11. Xing Fang, Jaejin Lee, and Samuel P. Midkiff. An optimizing and retargetable fence insertion algorithm. Technical Report ECE-HPCLab-033002, High Performance Computing Lab, School of Electrical and Computer Engineering, Purdue University, 2003.
12. Stephen Hartley. *Concurrent Programming: the Java Programming Language.* Oxford University Press, 1998.
13. Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, August 1998.
14. Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38:139–144, 1996.
15. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

16. Doug Lea. *Concurrent Programming in Java.* Addison Wesley, 1999. URL: `http://gee.cs.oswego.edu/dl/cpj`.
17. Douglas Lea and Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
18. Erik Ruf. Effective synchronization removal for java. In *Conference on Programming Languages, Design, and Implementation (PLDI)*, 2000.
19. C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proc. of the 14th Annual Int'l Symp. on Computer Architecture (ISCA'87)*, pages 234–243, 1987.
20. Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.
21. Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago IL, 2005.
22. Zehra N. Sura. *Analyzing Threads for Shared Memory Consistency.* PhD thesis, University of Illinois at Urbana-Champaign, 2004.
23. Chi-Leung Wong. *Thread Escape Analysis for a Memory Consistency Model-aware Compiler.* PhD thesis, University of Illinois at Urbana-Champaign, 2005.