Chapter 1

EFFECTIVE CACHING FOR OBJECT ORIENTED ENVIRONMENTS

Dan Page
University Of Bristol
page@cs.bris.ac.uk

Henk L. Muller
University Of Bristol
henkm@cs.bris.ac.uk

James Irwin
University Of Bristol
jimbob@cs.bris.ac.uk

David May
University Of Bristol
dave@cs.bris.ac.uk

Abstract Conventional cache memories act to bridge the gap in speeds between the processor and main memory. However, typical cache hardware takes no account of the characteristics of specific programs and may incur a performance penalty because of such an approach. We propose an architecture which gives the compiler control of a cache that can be split into protected partitions. In this paper we discuss how such a device can be used in an object oriented environment to eliminate cache interference and maximise performance. These features are especially useful in the field of real-time programming, where cache determinism is a limiting factor in performance. Additionally, the small size and resulting low thermal and power profiles of the cache are valuable in embedded devices such as smart-cards and PDA systems.

Keywords: Java, cache partitioning, high performance, predictability, determinism

1. Introduction

Many computer architectures rely heavily on the use of cache memory to enable their processing units to operate at high speed. However, conventional cache hardware takes no account of the characteristics of specific programs and may incur a performance penalty by optimising for the average case. Furthermore, the introduction of various hardware devices [23, 11, 22] that seek to improve overall performance has decreased both the determinism and predictability of modern cache systems.

Object oriented systems may react badly to these features of cache memory since their memory access patterns are often atypical when compared to those used to empirically guide cache design. Although specific architectures [26] have addressed the problem of performance, they may still pay a penalty through a lack of flexibility and do little to ease the lack of cache determinism and predictability. The resulting situation is that embedded [24] and real-time [4] object oriented systems may not perform as expecting when using conventionally designed, modern cache technology.

To address these issues, we propose to replace the conventional cache architecture with a partitioned cache [18, 12]. A partitioned cache is a direct-mapped-like cache that can be dynamically partitioned into protected regions through the use of specialised cache management instructions. The configuration of the cache and mechanism to determine which partition is used for a given memory-related instruction is achieved by alteration and addition to the instruction set architecture [16] (ISA). Unlike conventional caches, the partitioned cache is therefore visible to software running on the host processor. This allows the compiler and operating system (OS) to utilise the cache management instructions and load/store mechanism and allocate partitions of the cache to specific data objects and streams of instructions. This in turn offers control of data and instruction persistence to the compiler or OS and eliminates interference which may lead to non-deterministic behaviour. This can be done on a per-application basis thus eliminating the average case optimisation found in conventional systems.

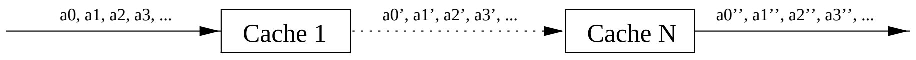


Figure 1.1. Sequential caches filter memory references.

An interesting way of describing what the partitioned cache is trying to achieve is to use the concept, proposed by McKee et al. [25], of *optical filters* as a metaphor for how caches work. Each level of a conventional cache hierarchy acts as a filter that translates an input stream of memory references into an

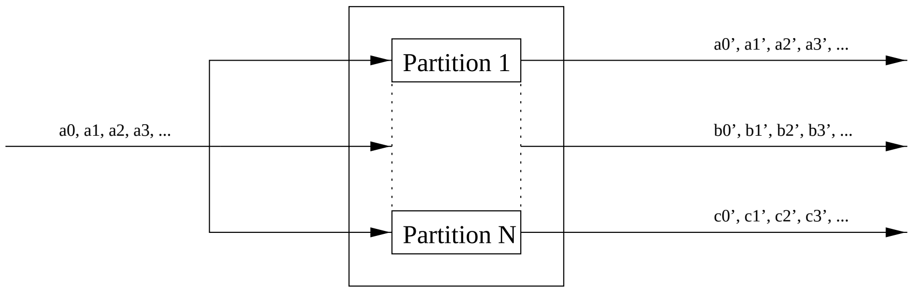


Figure 1.2. A partitioned acts to defract the reference stream.

output stream dependent on the properties of the cache at that level. The hope is to combine the filters so that they remove as many references as possible, reducing the load on slower parts of the memory hierarchy and maximising performance. In this context, a partitioned cache acts as a *defraction lens* by separating the stream of input references into a number of sub-streams. These sub-streams are passed through partitions in the cache which are configured to address the specific features of the associated stream of references.

Research with non-object oriented, C style languages [18, 16] shows that by performing suitable analysis at compile time, the process of partition allocation and management can be automated and produce increased performance and determinism, often with a decrease in required cache size. This paper demonstrates how an object oriented language such as Java can benefit from a similar approach and presents some ideas as to how such an approach may be realised in future research.

2. Java

Object oriented environments relate well to the methodology employed by a partitioned cache. Encapsulating instructions and data into classes and protected objects is mirrored well by the segregation of accesses to those objects using cache partitions. It is therefore a natural step to move our research from a C style language to an object oriented alternative. Java is a suitable language to study for a number of reasons:

- Java's rich class format lends itself well to holding additional semantic information such as partition identifiers. By holding the partition requirements of each class as fields of that class, a hardware based Java Virtual Machine (JVM) can manage the configuration of the cache as objects are created and destroyed or via the class loading mechanism.

- The feasibility of real-time and embedded Java systems is a problem. The use of a partitioned cache enhances cache determinism and predictability and reduces cache footprint, resulting in lower power consumption, which will improve the chances of such systems succeeding.

- HotSpot [2] related technologies have already demonstrated that the JVM lends its self well to adapting the run-time environment to suit application behaviour - a technique fundamental to the dynamic management of a partitioned cache.

- The lack of pointers, and associated pointer arithmetic, in Java eases some of the data partitioning problems introduced by object aliasing [29]. With no such aliasing problems, any reference to an object will always use the partitioning information held by that object and not via a reference to the object. This removes the problem of inconsistencies in a view of the data.

Although we have concentrated on a JVM style environment, many of the ideas apply equally as well to situations where code may be altered by for example just-in-time (JIT) compilation. This sort of native code will exhibit similar memory access patterns as interpreted code any will benefit in similar ways from the segregation imposed by cache partitioning.

3. Data Cache Partitioning

Previous work [18, 16] has demonstrated techniques for dealing with scalar and vector data, as well as accesses to the stack, in such a way that interference between objects is eliminated and performance is maximised. We expect many of the same ideas to be applicable to an object oriented environment. Several possibilities exist for harnessing the flexibility of a partitioned cache to store data objects:

- Utilise current segregation strategies [21] to allocate data objects to cache partitions based on their reference behaviour and usage lifetime.

- With the average object typically being around 10 words in size, and there being more small objects than large ones, it may prove worthwhile partitioning the cache on a per-object basis. This would ensure that no object could ever interfere with any other. Another approach would be to partition accesses based on the class of the object. This would result in less of an impact on cache resources at the cost of an increase in interference. It is important to note that both these strategies are complementary to other data placement [5] optimisations and object grouping techniques at other levels of the memory hierarchy [27].

- For environments in which generational garbage collection is employed, it would be feasible to partition the cache according to the number of generations [9] and steps within each generation as well as the allocation space for large objects. This effectively produces a configurable degree of associativity which has been shown [28, 31] to be effective in reducing conflicts between different generations of objects. Furthermore, this method of partition allocation is complementary to other optimisations [6] that deal with cache consciousness.

Other uses of cache partitions, provided that the analysis is possible, include eliminating heap usage for transient objects by allocating them in cache partitions. Current compilers use registers for temporary values but can not fit larger objects into this type of fast storage and thus revert to storing them in main memory. By moving the storage of transient objects one level up the memory hierarchy into the cache, we can potentially reduce cache pollution introduced by these objects and significantly decrease the number of accesses to heap memory. In turn, this would speed the process of garbage collection by culling a great deal of objects from the search space. By eliminating the need to allocate backing memory for the transient cache partition, we also reduce need for costly cache flushes to main memory. The drawbacks of this technique are the requirement for complex cache management systems that could impact on cache performance. Specifically, with partitions being allocated and reused quickly, the problem of fragmentation, which is not considered here, needs to be dealt with in a cost effective manner.

To illustrate how the more simple of these schemes might work, consider the implementation of the *daxpy* kernel, from the Linpack [7] benchmark suite, shown in Figure 1.3. The diagram in Figure 1.4a shows how the cache may look having been partitioned on a per-object basis by the partitioning information from the data objects in the *daxpy* kernel. The access to objects through correct partitions could be achieved by arranging for the memory management mechanism to honour the partition identifiers of each object. We can see that partitions have been allocated for $args$, dx and dy which will act to prevent interference between the objects. Prefetching can be performed for each object safe in the knowledge that prefetched data has a high probability of being used before eviction. The partitions can be sized according to the persistence requirements of each object. In this case, we are simply streaming through the arrays dx and dy and so single line partitions would suffice. Partitions have also been allocated for scalar objects n, i and da which can be packed into a partition sized to hold them all. Finally, a dedicated stack partition has been allocated so that intermediate calculations which use the stack will not interfere with other data objects.

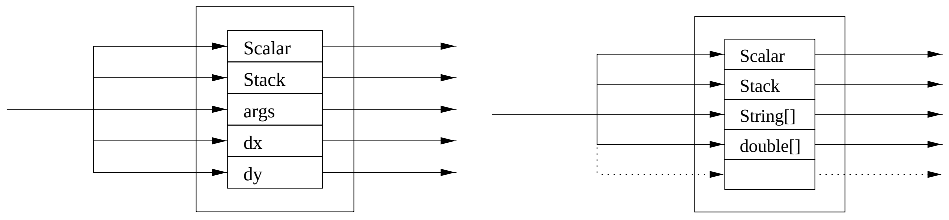
Figure 1.4b shows how the cache may look having been partitioned for data objects using a per-class strategy. In this case, the class loading mechanism

```
public class daxpy
{
  public static void main( String[] args )
  {
    double[] dx = new double[ 64 ];
    double[] dy = new double[ 64 ];
    int n       = 64;
    double da   = 0.6;

    daxpy( dx, dy, da, n );
  }

  public static void daxpy( double[] dx, double[] dy, double da, int n )
  {
    for( int i = 0; i < n; i++ )
    {
      dy[ i ] += da * dx[ i ];
    }
  }
}
```

Figure 1.3. An example implementation of the *daxpy* kernel.



(a) Per-object partitioning (b) Per-class partitioning

Figure 1.4. Per-object and per-class partitioning for *daxpy*.

is altered to configure the cache according to the partitioning information inserted into each class by the compiler analysis phases. We can see that the dx and dy will be accessed through the same partition because they are of the same type. Although this *may* re-introduce interference between the objects as described above, we have reduced the resource requirements even for this small program. As program size and complexity increases the number of objects will also increase but this per-class strategy will still manage to cull the large number of partitions required by per-object partitioning.

It is intuitively clear that neither of these techniques alone will yield results which compare favourably with those from compiler directed techniques. Knowledge about the usage patterns of the objects and classes, which can only be extracted from the application source code, will always be an advantage in the allocation of cache partitions. In conclusion, it would appear that a combination of partitioning techniques would be suitable to cache data objects. It is easy to envisage a system whereby large objects are allocated their own partitions, smaller objects are partitioned on a generational basis and stack accesses are routed through another partition. The flexibility of the partitioned cache allows this level of segregation that, provided suitable compiler analysis is conducted on the types of usage for each object, will result in more predictable cache behaviour, better cache utilisation and potentially higher performance. However, a compromise in the overhead, in terms of processor cycles, introduced by complex cache management verses performance gains must be reached for such a scheme to be feasible.

4. Instruction Cache Partitioning

In general, instructions that are not resident in the cache of a Java processor will result in a performance penalty by forcing an access to main memory. In an object oriented environment, the large number of classes and methods can reduce the locality of instruction references and introduce interference that in turn may defeat conventional caching strategies. Multi-threading may be used to hide this latency to some extent but can act to further pollute the instruction cache by introducing inter-thread interference.

Java classes, especially those influenced by the JavaBean style of programming, tend to be made up of small methods. Studies [15] demonstrate that there are on average six methods in each class and that a class is made up from an average of thirty bytecodes. With this in mind it would be easy to place the bytecodes in memory without consideration for the impact on cache performance. For example, with a large number of classes resident in memory, the class loading mechanism may place the bytecodes for different methods so that they map to the same space in the cache. With method calls within Java being measured as often as every four bytecodes, it is easy to envisage a situation where methods are constantly evicting each other, effectively thrashing the cache.

Techniques already exist to reorganise a process image [8, 19] in order to improve the performance of an instruction cache. However, these methods are reliant on profiling and are constrained in that they need to consider the whole application at once. In a dynamic environment such as that provided by Java, new and potentially unseen code can be loaded and linked at run-time which would render this system less effective.

We propose to use the partitioned cache to access the bytecode instruction streams for different classes and, in some cases, methods, through separate partitions. By introducing separation of the overall instruction stream, we can en-

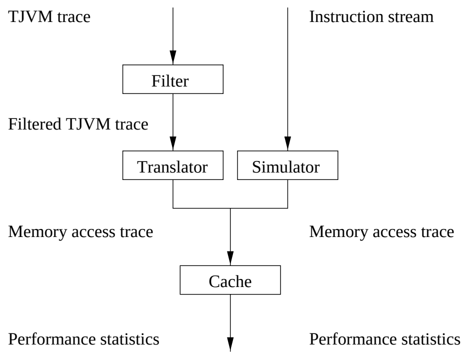


Figure 1.5. Trace translation and execution tools.

sure that inter-class and inter-method interference is eliminated in cases where it would be most costly. Studies have shown that average size of a Java class is roughly 3000 bytes. For classes smaller than this, it would be feasible to allocate a partition to each class sized to accommodate the whole class. For other classes, it would be more productive to identify individual methods as being suitable for their own partitions so as to limit the overall cache resources used. As an alternative to both strategies, it is also possible to allow the programmer to mark critical, or real-time, sections of source code so that the compiler can more easily infer the correct partitioning strategy.

5. Results

In order to obtain some idea as to the effectiveness of these techniques, we constructed tool, shown in Figure 1.5, which reverse engineer a stream of partitioned memory references from a trace generated by the Tracing JVM [30] (TJVM).

The first stage of processing is to filter the trace so that only events relevant to the program under investigation remain. For example, it may be valuable to exclude the events generated by the boot sequence of the TJVM so that only the application or kernel under investigation is considered. By excluding events of this type, we are effectively assuming that the JVM is *warmed up* before the application starts and that no other background operations are required.

After filtering has been conducted, the translator passes over the trace to extract all the object, class and stack information. This information is used to guide the creation of partitions, to deal with accesses to data objects and the stack, and to create a memory map of the objects in use. Objects are assigned addresses in virtual memory in the order that they are created, without regard for garbage collection, with the stack starting after the last object. With the partitioning and layout information in place, a second pass of the trace translates relevant trace events into memory accesses to the virtual memory. At this point, the translator also injects instructions required to manage the cache so it is suitably configured for the memory access trace. The resulting stream of cache access and management instructions is then fed to our existing memory hierarchy simulation tools to measure the effectiveness of the partitioning strategies under investigation.

Although this technique is far from ideal, it allows us to demonstrate the effectiveness of the idea without the need for the implementation of a partitioned cache aware Java compiler and run-time environment. The development of such a system was beyond the scope of our project in terms of the time and resources required. Not using a more complete system will inevitably yield worse results than the compiler-directed equivalent discussed elsewhere but we were able to produce interesting results even when constrained by two major caveats:

- We ignore events generated by accesses to the stack which can be effectively and easily cached using a small buffer [3] or private cache partition. The absence of these events allows a clearer picture of the other activities under investigation.
- The tools are currently only able to analyse the requirements for caching data objects due to a lack of support for bytecode related events in the TJVM. All results therefore relate only to the caching of data objects in the benchmarks under investigation.

By using the discussed partitioning system, the *daxpy* kernel shown in Figure 1.3 performs better on a partitioned cache than similar sized conventional caches. When a per-object partitioning strategy is employed, we create five small partitions totalling eight cache lines. This partitioning results in halving the number of cache misses, as shown in Figure 1.6, when compared to similar sized direct-mapped caches. Only when the direct-mapped cache is allowed enough space to contain the working set does it attain equivalent performance.

Obviously *daxpy* is a trivial example and bears little relation to real world situations. To more rigorously test the effectiveness of a partitioned cache under more varied partitioning schemes, we employed the trace translation system on a number of more substantial sequential benchmark programs. We re-programmed simple multi-media algorithms from our own benchmarks in Java and added other, more numerically oriented codes from the Linpack [7], SciMark [20] and Java Grande [14] suites. These benchmarks provide a good mix of non-object oriented programming written in an object oriented language and true object oriented applications which reflect the diverse way in which Java is used.

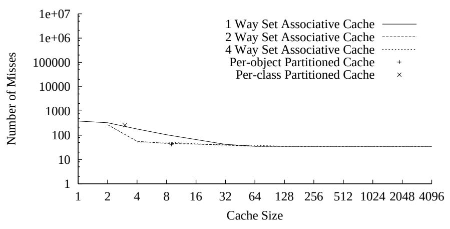


Figure 1.6. Cache performance during execution of the *daxpy* kernel.

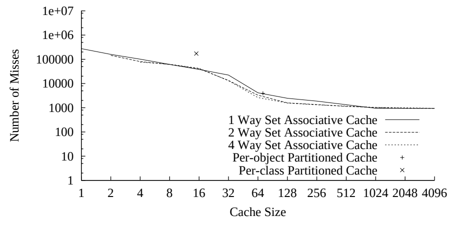
In addition to this we examined the Richards kernel, a reasonably sized benchmark program which simulates the task dispatcher in the kernel of an operating system. The fact that it simulates a real-world, performance sensitive problem makes it an interesting example to test with our partitioning mechanisms. Two different translations of the kernel, from Jonathan Gibbons and L. Peter Deutsch, are used. The Gibbons translation utilises less object-oriented features of the Java language than the Deutsch version which has been shown to produce noticeably different performance profiles.

5.1 Single-threaded Benchmarks

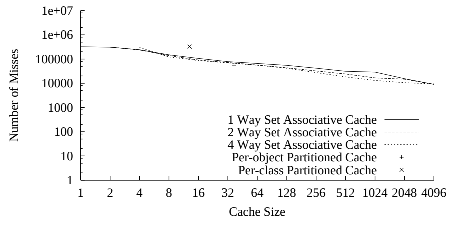
The first stage of testing treats each benchmark as the only thread running on the JVM. This effectively rules out inter-thread interference and instead measures the levels of intra-thread or self interference. Comparisons between results for a partitioned cache, after performing per-object and per-class partitioning, and conventional set-associative caches are shown in Figures 1.7 and 1.8. These graphs show the number of cache misses that occur during the run-time of the benchmark against the size of the cache.

These benchmarks reveal a number of trends that run throughout the results. The major point of note is that the partitioned caches rarely perform better than similar sized conventional caches due to several factors :

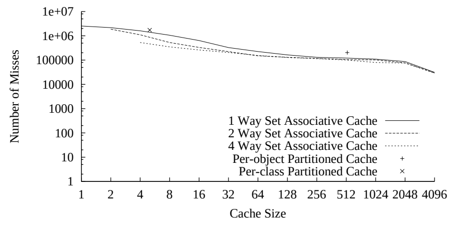
- Allocating even small partitions on a per-object basis is costly, in terms of the resources required, without some sort of partition reuse strategy. This is most strongly highlighted in the Richards benchmarks where the number of objects and partitions is very high even though the partition allocated to one object can often be reused by another with a non-overlapping lifespan. This technique is not available to us using the current tool-chain and the partitioned caches are therefore often much larger than they need to be. In addition to this, partitions allocated for objects often have some slack space as a result of the requirement that partitions be a power-of-two in size. With the average size of a Java object at



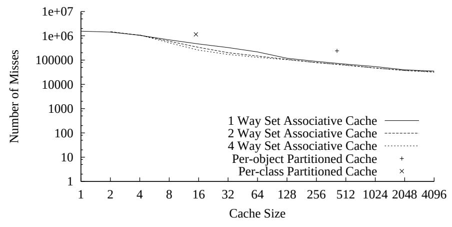
(a) CCITT G.723 Audio Decoder.



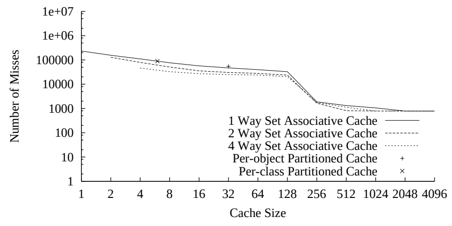
(b) GIF Image Decoder.



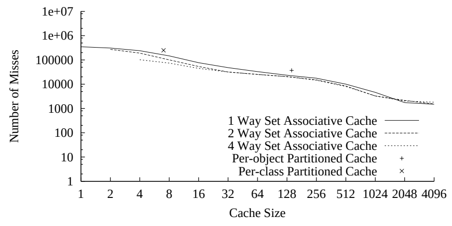
(c) Linpack.



(d) MPEG Video Player.



(e) Fast Fourier Transform.



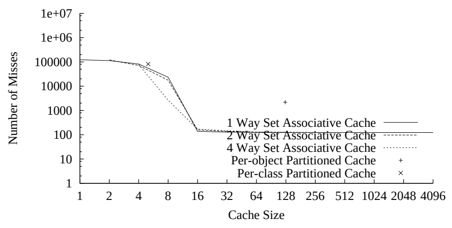
(f) Dense LU Matrix Factorization.

Figure 1.7. Per-object and per-class partitioning compared to conventional caches for a variety of single-threaded benchmarks (1).

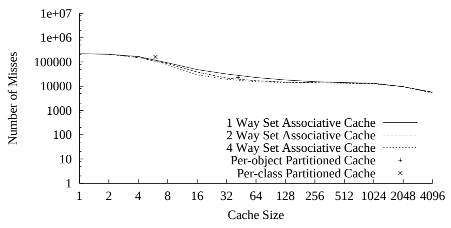
around 10 words, this can result in a large amount of redundant space, in relation to the used space, when considering the entire benchmark.

- Using a per-class strategy for partition allocation is effective in reducing the resources used by each benchmark but suffers from poor performance. Furthermore, per-class partitioning re-introduces much of the unpredictability of conventional cache systems by allowing objects to interfere with each other depending on their class.

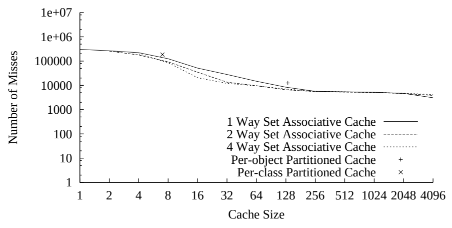
- The lack of compiler directed, source language analysis results in an absence of a number of partitioning features that would otherwise help to improve performance. Two of these features are the ability to correctly configure the stride parameter of a partition, hence increasing the amount of useful data in the partition, and the ability to assign more than one partition per data object. Both of these features require that analysis is performed on the source language to specialise the cache to the



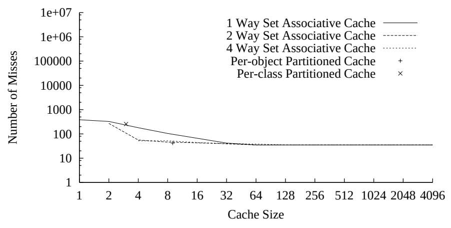
(g) Monte Carlo Integration.



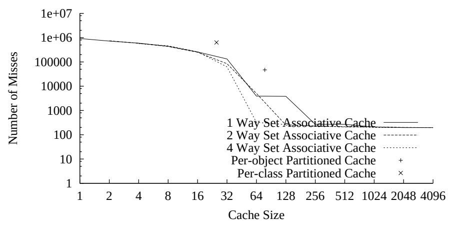
(h) Sparse Matrix Multiply.



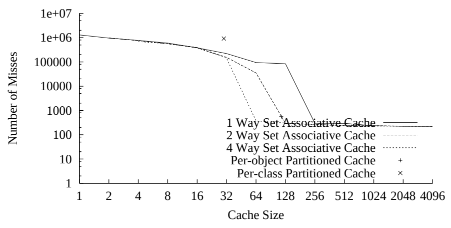
(i) Jacobi Successive Over-Relaxation.



(j) daxpy.



(k) Richards [Gibbons].



(l) Richards [Deutsch].

Figure 1.8. Per-object and per-class partitioning compared to conventional caches for a variety of single-threaded benchmarks (2).

application requirements. The lack of the ability in our trace driven partitioning system underlines the value of such analysis in improving the overall performance of a partitioned cache.

- The small cache size required to hold the working data set of some benchmarks is an example of optimisation for the average case. In, for example, the Richards benchmarks, a conventional cache of only 256 lines in size is enough to attain near optimal performance. However, modern processors regularly contain caches which are many times this size, wasting resources that may be better spent elsewhere.

One interesting feature of the results from the Richards benchmarks are that the more object-oriented version, the Deutsch benchmark, performs significantly better when a per-object partitioning strategy is used than the Gibbons benchmark which makes less use of object-oriented design. Additionally, the conventional set-associative caches perform worse on the object-oriented Deutsch benchmark than the Gibbons version. This is perhaps a hint at the amount of inter-object interference that exists in the benchmark and how using a partitioning strategy can help to prevent it.

5.1.1 Partition Reuse. To demonstrate the advantage that reusing partitions can offer, we constructed an experiment where the partitioning was implemented by hand. We chose the Linpack [7] benchmark in which a large number of array objects are created and used. By examining the source code for the benchmark in the same way a compiler might, we were easily able to deduce that the vast proportion of the arrays did not overlap in terms of lifespan and could therefore share partitions. This kind of operation may also be performed in conjunction with garbage collection whereby the partition for an object that has been collected may be reused by subsequently allocated objects.

It is interesting to note that the graphs of performance in Figures 1.7 and 1.8, show it is generally much harder to improve the partitioned cache by saving size than it is improving performance. The size of the partitioned cache needs to reduced by many times, moving the performance point to the left on the graph, in order to gain an advantage over the conventional caches while improving performance, moving the point down on the graph, quickly yields better comparisons. However, reducing the total cache size used is easier in our system than improving performance. This is because better understanding of the application reference behaviour needed to improve performance is not possible due to the lack of compiler based source code analysis.

By implementing a suitable partition configurations in the benchmark trace, we produced the results in Figure 1.9 which show a comparison between the cache performance with and without partition reuse. When partition reuse is introduced, it has no effect on the performance of the cache. This is expected as the shared partitions are used in a mutually exclusive manner by the array accesses and hence produce the same number of misses as if they were housed in different partitions. However, the reuse strategy reduces the number of partitions used from 209 to 10 and the number of lines used from 523 to 29. This represents a massive saving in resources and results in the partitioned cache comparing more favourably to similar sized conventional caches.

By considering the fact that the same scheme could be applied to all the benchmarks in our original experiments with differing degrees of resource saving, the originally disappointing results start to look more encouraging. It should also be noted that partition reuse does not affect the predictability of the cache in any way and that it could easily be implemented in a Java based environment by a combination of the compiler and elements of the virtual machine.

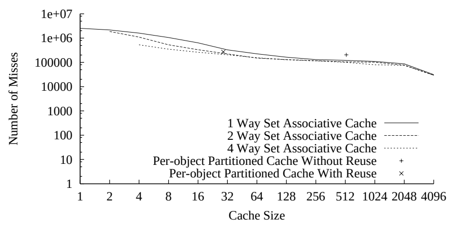


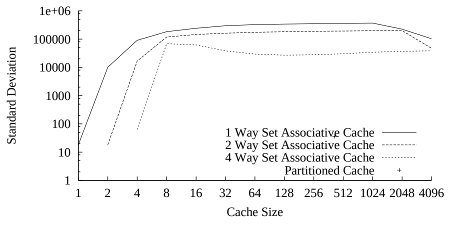
Figure 1.9. A comparison of how partition reuse effects cache performance.

5.1.2 Determinism and Predictability. It is important to remember that performance is not the only factor that needs to be considered. While the performance of the partitioned cache may not have seen great benefits when compared to conventional caches, the increased determinism of the cache is undeniable.

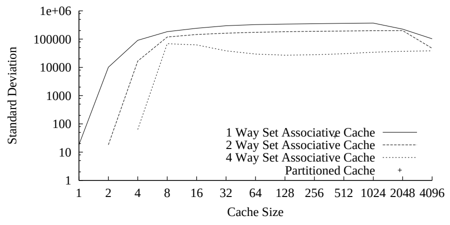
The performance of the conventional caches is a product of the layout of objects in memory and the configuration of the cache. The scheme used to layout objects in memory is simplistic and produces the same layout each time it is used on the same TJVM trace but this is not always the case in real life situations. The address at which each object is placed by the layout mechanism in a real JVM is affected by a number of factors such as the heap location in real memory and the operation of the garbage collector. If each time the application is run the objects are placed at a different addresses, it could be the case that a conventional cache will have a different and unpredictable performance profile. Because of the segregation imposed by the partitioned cache, the movement of objects in memory between runs does not affect the cache in the same way.

We were able to prove this fact by altering the way objects are allocated on the heap between benchmarking runs of the MPEG player. Figure 1.10 shows the results of calculating the standard deviation of the set of allocations for a variety of cache types thus hinting at how the performance of the benchmark set varies on each cache. These results show that while the partitioned cache performs slightly worse in most cases than conventional caches, it is far more deterministic over changes of the allocation strategy while the variation in conventional caches is significantly larger. This difference is a direct result of the protection from interference offered by the partitioned cache. With no interference, the movement in memory of an object can never influence the cache space allocated to another object so changing the allocation strategy has no impact on the performance of each partition.

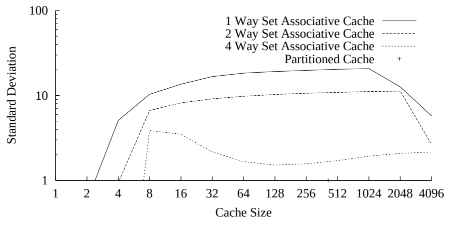
By considering the fact that the object allocation strategy can have dramatic effects on the performance of conventional caches, the results from our original experiments should be more encouraging. In all cases of allocation strategy,



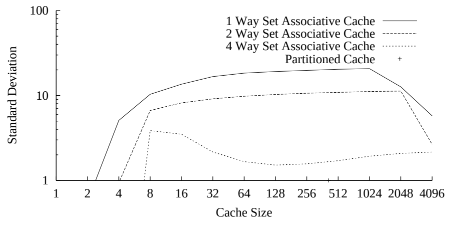
(a) Hits.



(b) Misses.



(c) Hit Ratio.



(d) Miss Ratio.

Figure 1.10. MPEG Video Player performance variation, measured using standard deviation, as a result of differing object allocation.

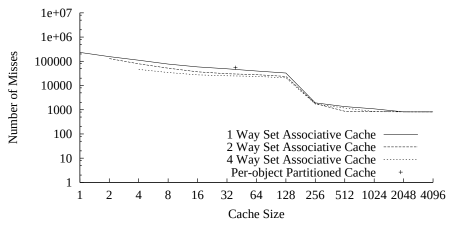
the partitioned caches will perform the same way, their deviation being zero in all cases and hence not visible on the graphs. In situations such as real-time environments this is as valuable a property as average case performance.

5.2 Multi-threaded Benchmarks

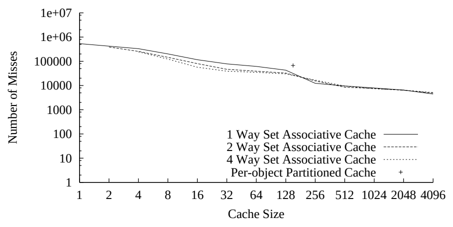
To complement the experiments from Section 5.1 in which each benchmark was treated as the only running thread, we devised some tests in which combinations of benchmarks are run at the same time to create composite, multi-threaded benchmarks.

Each thread in a multi-threaded benchmark is one of either Fast Fourier Transform, Jacobi Successive Over-Relaxation, Sparse Matrix Multiply or Dense LU Matrix Factorization which are taken from the SciMark [20] benchmark suite and have comparable run-times. These kernels are added, in round-robin order, to produce successively more multi-threaded situations which are allowed to run under the normal thread-scheduling system present in the host JVM. The context switching between threads acts to introduce inter-thread interference into the cache access profile for each benchmark.

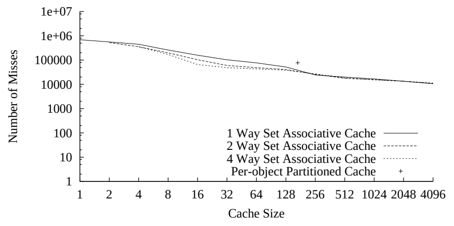
By looking at the performance graphs of per-object partitioned caches against conventional set-associative caches shown in Figure 1.11, we can see that the impact of running the benchmarks in a multi-threaded style is minimal. Most obviously, there isn't the dramatic decrease in performance we found from previous work [17]. This is due to two reasons. Firstly, it is clear by looking at



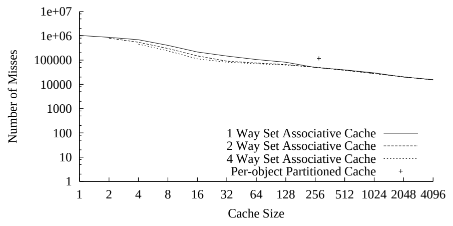
(a) 1 SciMark thread(s).



(b) 2 SciMark thread(s).



(c) 3 SciMark thread(s).



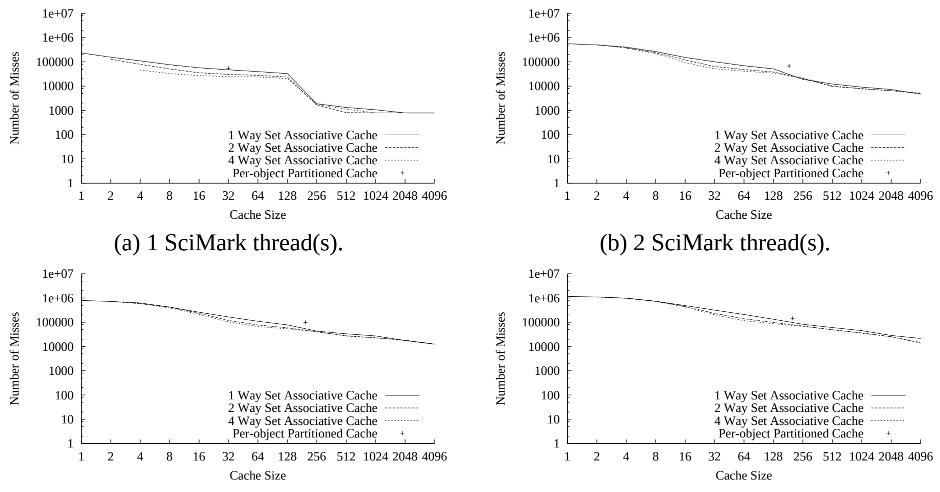
(d) 4 SciMark thread(s).

Figure 1.11. Per-object partitioning compared to conventional caches for a variety of course-grain multi-threaded benchmarks.

the traces that Java thread scheduling mechanism doesn't create enough fine-grained parallelism to create the types of interference seen previously. More specifically, the time quantum for each thread is long enough so the cost of re-populating the cache with useful data on a context switch is small compared to the cost of long spells of cache use from a given thread. Secondly, because of the simplistic object allocation mechanism used in this example, it is possible that the objects are aligned in the cache in a manner which prevents them interfering with each other as much as could be possible. This is the same situation as was demonstrated with single-threaded benchmarks in the previous section.

5.2.1 Scheduling Strategy. To better simulate a more fine-grained form of thread switching, such as that found in picoJava [1] based hardware thread systems, we performed a further phase of translation to each benchmark trace. This phase of translation takes instructions from all threads and produces a resultant trace in which the threads are switched between far more often than in a software based threading system. The resultant trace more closely mimics the kinds of behaviour of a multi-threaded architecture [13] as it uses hardware based thread contexts to mask memory access latency. The results from these fine-grained versions of the multi-threaded benchmarks are shown in Figure 1.12.

Although the performance of conventional caches is slightly decreased by the introduction of a fine-grained scheduling strategy, the performance of the



(a) 1 SciMark thread(s).

(b) 2 SciMark thread(s).

(c) 3 SciMark thread(s).

(d) 4 SciMark thread(s).

Figure 1.12. Per-object partitioning compared to conventional caches for a variety of fine-grain multi-threaded benchmarks.

partitioned cache remains the same. The amount of impact that changing the scheduling strategy has on the performance of the set-associative caches is disappointingly small, it further underlines the value of the determinism demonstrated by the partitioned cache.

6. Conclusions

In this paper we have worked at performing object-oriented cache partitioning without the use of compiler based source code analysis. This has forced the investigation of not only aspects of object-oriented programs but how a runtime system can implement some level of cache partitioning without compiler support. In this respect it is closely related to work on adaptive caches [10].

The results of this work have been underwhelming and highlight the need for, and value of, good compiler based analysis to guide the configuration of a partitioned cache. It is imperative to note that many of the problems in the results from Sections 5.1 and 5.2 are as a direct result of the lack of this compiler analysis which doesn't allows us to realise the improvements in performance which might be intuitively expected. Furthermore, the lack of any source code analysis forces us to accept a compromise between a cache configuration specialised to the application and a configuration for the average case application we were trying to avoid. Additional penalties to our results arise from the effects of class-loading and other background activities which, without further development of the TJVM and our own tools, act to cloud the understanding of the benchmarks under investigation.

However, the results from experimentation in this differing environment have yielded some positive ideas. Firstly, it is clear that the concepts of an object-based memory hierarchy and a partitioned cache fit very well together. The scope for further work to exploit this relationship is considerable and we expect the results to be encouraging. Secondly, the results which compare two differing implementations of the Richards kernel hint that there is some performance advantage in using a partitioned cache on object-oriented programs. Conventional caches perform worse in this sort of environment than when using the non-object-oriented application which is sure to impact more significantly as program size and complexity increases. Both these results, coupled with the improved determinism and predictability qualities of the cache, mean that real-time and embedded Java products can be more easily realised. Specifically, by using suitable compiler based tools designers can know at compile time what the cache requirements of their application is and be sure that the performance of the cache will not be influenced by other software running on the same processor.

In terms of the partitioned strategy used, it is clear that there must be some compromise between the per-object, full object protection style and the per-class, resource use minimisation style. While using more complex benchmark applications, such as SpecJVM98, may be able to suggest a suitable technique, the idea of a garbage collector generation [28, 31] partitioning is attractive. This system would tightly couple the memory hierarchy into a cohesive entity capable of performing the kind of transient object allocation discussed in Section 3 and reaping massive rewards from doing so. The implementation of this kind of multi-layered partitioning approach is the subject of current research which we expect to yield very positive results.

References

[1] picoJava II Processor Core Description. Technical Report 805-4634-01, Sun Microsystems, April 1999.

[2] The Java Hotspot Performance Engine Architecture. Technical report, Javasoft, April 1999.

[3] C. Bailey. *Optimisation Techniques for Stack-Based Processors*. PhD thesis, Department of Electrical and Electronic Engineering, University of Teesside, July 1996.

[4] Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, January 2000.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[6] T.M. Chilimbi and J.R. Larus. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *Internation Symposium on Memory Management*, October 1998.

[7] J.J. Dongarra. Performance of Various Computers Using Standard Linear Equation Software. Technical Report CS-89-85, Department of Computer Science, University of Tennessee, June 2000.

[8] N. Gloy, T. Blockwell, M.D. Smith, and B. Calder. Procedure Placement Using Temporal Ordering Information. In *30th International Symposium on Microarchitecture*, December 1997.

[9] E.G. Hallnor and S.K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *27th International Symposium on Computer Architecture*, June 2000.

[10] T.L. Johnson and W.W. Hwu. Run-time Adaptive Cache Heirarchy Management via Reference Analysis. In *24th International Symposium on Computer Architecture*, pages 315–326, June 1997.

[11] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffer. In *17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[12] T. Juan, D. Royo, and J.J. Navarro. Dynamic Cache Splitting. *15th International Conference of the Chilean Computational Society*, 1995.

[13] J.S. Kowalik. *Parallel MIMD Computation: The HEP Supercomputer and its Applications*. MIT Press, 1985.

[14] J.A. Mathew, P.D. Coddington, and K.A. Hawick. Analysis and Development of Java Grande Benchmarks. In *Procedings of the ACM 1999 Java Grande Conference*, June 1999.

[15] Markus Mohnen. Private communications regarding results of the JOPT Java optimisation project.

[16] D. Page. *Effective Use of Partitioned Cache Memories*. PhD thesis, Department of Computer Science, University of Bristol, 2001.

[17] D. Page, J. Irwin, H.L. Muller, and D. May. Effective Caching for Multithreaded Processors. In *Communicating Process Architectures 2000*, pages 145–154. IOS Press, September 2000.

[18] D. Page, D. May, J. Irwin, and H.L. Muller. Microcaches. In *6th International Conference On High Performance Computing*, pages 21–27. Springer-Verlag, December 1999.

[19] K. Pettis and R.C. Hansen. Profile Guided Code Positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1990.

[20] R. Pozo and B. Miller. SciMark 2.0. `http://math.nist.gov/scimark2/`.

[21] M.L. Seidl and B.G. Zorn. Segregating Heap Objects by Reference Behaviour and Lifetime. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[22] A. Seznec and F. Bodin. Skewed-Associative Caches. In *Parallel Architectures and Languages Europe*, pages 305–316. Springer Verlag, July 1993.

[23] A.J. Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. *IEEE Transactions on Software Engineering*, March 1978.

[24] Sun Microsystems. Technical Overview of EmbeddedJava Technology. `http://java.sun.com/products/embeddedjava/overview.html`, August 2000.

[25] D.A.B. Weikle, S.A. McKee, and W.A. Wulf. Caches As Filters: A New Approach to Cache Analysis. *6th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1998.

[26] I. Williams and M. Wolczko. An Object-Based Memory Architecture. In *4th International Workshop on Persistent Object Systems*, 1991.

[27] I. Williams, M. Wolczko, and T. Hopkins. Dynamic Grouping in an Object Oriented Virtual Memory Heirarchy. In *European Conference on Object-Oriented Programming*, pages 79–88. Springer-Verlag, 1987.

[28] P.R. Wilson, M.S. Lam, and T.G. Moher. Caching Considerations for Generational Garbage Collection. In *ACM Conference on Lisp and Functional Programming*, pages 32–42, June 1992.

[29] R.P. Wilson. *Efficient Context-Sensitive Pointer Analysis For C Programs*. PhD thesis, Computer Systems Laboratory, Stanford University, 1997.

[30] M. Wolczko. Using a Tracing Java Virtual Machine to Gather Data on the Behaviour of Java Programs. Technical report, Sun Microsystems, March 1999.

[31] B. Zorn. The Effect of Garbage Collection on Cache Performance. Technical Report CU-CS-528-91, University of Colorado, May 1991.