Thread-local Scope Caching for Real-time Java

Andy Wellings
Department of Computer Science
University of York, UK
andy@cs.york.ac.uk

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Abstract

There is increasing convergence between the fields of parallel and embedded computing. The demand for more functionality in embedded devices means that complex multicore architectures will be used. In order to promote scalability and obtain predictability, on-chip processor-local private memory subsystems will be used. Whilst at the hardware level this is technical feasible, the more pressing problem is how such memory is presented to the programmer and how its local access is policed.

In this paper we illustrate how Java augmented by the Real-time Specification for Java can be used to present the abstraction of a thread-local scoped memory area. We show how to enforce access to the memory area to a single real-time thread. We implement the model on the JOP multiprocessor system and report on our experiences.

1 Introduction

Today's computers may have many different types of directly addressable memory available to them. Each type has its own characteristics [3] that determine whether it is

- volatile – whether it maintains its state when the power is turned off,
- writable – whether it can be written at all, written once or written many times and whether writing is under program control,
- erasable at the byte level – if the memory can be overwritten whether this is done at the byte level or whether whole sectors of the memory need to be erased,
- fast to access – both for reading and writing,
- expensive – in terms of cost per byte or power consumption.

Examples include dynamic and static random access memory (DRAM and SRAM), read-only memory (ROM and EPROM) and hybrid memory such as EEPROM (electrically erasable and programmable ROM) or FLASH memory. Memory may also be added and removed from the system dynamically. Furthermore, on-chip and off-chip caching may occur. The recent popularity of Field Programmable Gate Arrays (FPGAs) has added to this variety.

Individual computers are often targeted at a particular application domain, which will often dictate the cost and performance requirements, and therefore, the memory type used. For example, embedded computers targeted at, say, mass-produced consumer products will need to be cheap and, as a consequence, will only have a limited amount of fast (expensive) memory. In order to obtain maximum performance from the given computer, the programmer must make judicious use of the available memory. Placing a heavily used object in a slow memory area may seriously degrade the program's overall effectiveness, particularly if the system has a limited cache size or the cache has been turned off (or is not present at all) to ensure predictability.

As well as having different types of memory, many computers map input and output devices so that their registers can be accessed through memory location. Hence, some parts of the processor's address space will map to real memory and other parts will access devices. The situation may be further complicated because a Direct Memory Access (DMA) controller may access the real memory independently of the processor (although doing so may steal bus cycles from the processor). For multiprocessor systems, the above complexities are compounded by the possibility of local memory, shared memory or dual ported memory.

As we move towards more complex multicore architectures, the issue of safely programming these systems becomes paramount. The EU JEOPARD project is investigating the use of Java augmented by the Real-Time Specification for Java (RTSJ) as a base technology for programming these architectures for use in embedded applications.

1.1 Motivation and Scope

A pipelined processor architecture calls for high memory bandwidth. Chip Multiprocessor (CMP) systems that support

global shared memory increase the pressure on the memory subsystem. A standard technique to avoid processing bottlenecks due to the lower available memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis [5]. Furthermore, the difficulty of keeping multiple caches coherent limit the scalability of such systems.

In order to reduce the contention on shared memory subsystems, local memory subsystems can be created. The goal with such memory is to ensure that it is only accessed by one processor, and hence there is no need to cache its contents for access by other processors. Whilst at the hardware level this is technical feasible, the more pressing problem is how such memory is presented to the programmer and how its local access is policed.

1.2 Contributions

In this paper we address the problem of accessing private local memory in a CMP system. In particular, we consider how the facilities of Java (augmented by the Real-Time Specification for Java) can be used to define the notion of a *thread-local memory area*. We then detail the implementation of such an area on the JOP processor.

We assume that real-time threads are tied to a particular processor and do not migrate. This is often an assumption for schedulability analysis [20].

2 Memory Areas and the RTSJ

Memory areas were originally introduced into the RTSJ in order to extend the Java memory model so that it could provide access to non-heap data, and thus avoid the vagaries of garbage collection. However they also provide a mechanism by which a programmer can express locality of data: both to a schedulable object and to a physical region in memory.

The MemoryArea class is an abstract class from which all RTSJ memory areas are derived. When a particular memory area is entered, all object allocation is performed within that area. Using this abstract class, the RTSJ defines various kinds of memory including the following:

- HeapMemory – Heap memory allows objects to be allocated in the standard Java heap.
- ImmortalMemory – Immortal memory is shared among all threads in an application. Objects created in immortal memory are never subject to garbage collection delays and behave as if they are freed by the system only when the program terminates.
- ScopedMemory – Scoped memory is a memory area where objects with a well-defined lifetime can be allocated.

The ScopedMemory class is an abstract class that has several subclasses, including

- VTMemory – A subclass of ScopedMemory where allocations may take variable amounts of time.
- LTMemory – A subclass of ScopedMemory where allocations occur in linear time (that is, the time taken to allocate the object is directly proportional to the size of the object).

The memory used for allocated objects when a scoped memory area is active is called the scoped memory's backing store. It resides in a part of memory that is otherwise invisible to the application. It is separate from the memory required for the scoped memory object itself (which is allocated from the current memory area when the object is created). The backing store is usually assigned to the scoped memory object when the object is created, and it is freed when the object is finalized. When assigned, memory within the backing store used for allocated objects can be reclaimed when the scoped memory becomes inactive (that is, its reference count goes to zero).

Physical scoped memory areas allow the programmer to specify that the backing store should be created in memory with a particular characteristic (for example, shared memory) as well as the usual requirements for linear time allocation etc. Immortal physical memory is also provided.[1] The implementation of these physical memory classes assume the existence of a PhysicalMemoryManager class, which must be provided by the real-time Java virtual machine. This class can also make use of implementation-dependent classes, called filters, which help support and police the various memory categories. All these classes must support the PhysicalMemoryTypeFilter interface. The memory manager can also check that the program has the necessary security permissions before allowing access to physical memory. The classes are illustrated in Figure 1.

3 Thread-local Scoped Memory Area for Local Caching

In a Java and RTSJ context, local private memory must be viewed either as scoped or immortal memory. As, by definition, the Java heap is shared between all schedulable objects and Java threads. Only physical memory allows the programmer the ability to specify the location of the backing store. Only scoped memory allows fine control over the Java object graph to be policed – objects in heap or immortal memory cannot reference objects in scoped memory. *Hence, we use the notion of physical scoped memory areas as the programmers' model of a local read/write private memory*

[1] Note, the backing store for physical immortal memory is never reclaimed even if the associated object goes out of scope.

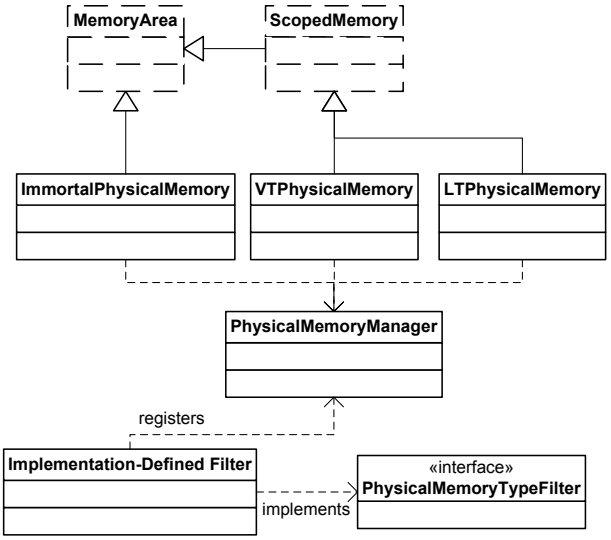

Figure 1. Physical Memory Related Classes

subsystem. To further ensure the locality of access we define the notion of a real-time thread-local scoped memory area. Such a memory area can only be entered by a single real-time thread and it can be held in a memory subsystem that is local to the processor executing the schedulable object. The goal is to implement such a model using the facilities of the RTSJ. We assume that RTSJ version 1.1 will support thread affinity and, therefore, allow real-time threads to be fixed to a processor and not migrate.

3.1 Thread-local Memory Areas

The PhysicalMemoryManager class of the RTSJ provides the interface between the application and the memory hierarchy of the machine on which the program is currently executing. Here we assume that the local private memory is called ON_CHIP_PRIVATE:[2]

```
public final class PhysicalMemoryManager {
  public static final Object ON_CHIP_PRIVATE;
  // The PhysicalMemoryManager knows about the
  // on-chip memory and keeps track of its use
}
```

The RTSJ physical scoped memory classes communicate with the memory manager when the programmer requests the creation of a physical scoped memory region. Here we will use the linear time version (LT):

```
public class LTPhysicalMemory {
  public LTPhysicalMemory(Object type, long size);
    // The constructor code communicates with
    // the PhysicalMemoryManager asking for
    // a memory chunk of the given size and type.
    // Throws various exceptions if the manager
    // can not oblige.

  public LTPhysicalMemory(Object type, long base,
                          long size);
    // The constructor code communicates with
    // the PhysicalMemoryManager asking for
    // a memory chunk of the given size and
    // type starting at a base address.
    // Throws various exceptions if the manager
    // can not oblige.

  public void enter(Runnable R);
  public void executeInArea(Runnable R);
  // ... other methods
}
```

In order to implement a local scope, it is important to ensure that only one thread can access it at a time. To meet this requirement, the obvious solution is to use Java's thread local data facility. The following illustrates how such a mechanism can be created.

```
// This class allows multiple local scopes to
// be created. The use of thread local data
// ensures that an SO can only enter its
// own scope.

public class ThreadLocalScope {
   public ThreadLocalScope(long size) {
     myMemory = new ThreadLocal<LTPhysicalMemory>();
     LTPhysicalMemory myMem =
       new LTPhysicalMemory(PhysicalMemoryManager.
                            ON_CHIP, size);
       // can throw out of memory
     myMemory.set(myMem);
   }

   public void enter(Runnable R) {
     if (myMemory.get() == null) {
       // throw exception, the calling thread
       // does not own this scope
     }
     else myMemory.get().enter(R);
   }

   public void executeInArea(Runnable R) {
     if (myMemory.get() == null) {
       // throw exception, the calling
       // thread does not own this scope
     }
     else myMemory.get().executeInArea(R);
   }

   // other needed scoped memory methods
   private ThreadLocal<LTPhysicalMemory> myMemory;
}
```

In essence, the above class is providing a wrapper around access to LTPhysicalMemory to ensure that a thread only accesses its own local ones. If a reference to one threads local scope escapes to another thread and that thread attempts

[2]Where private memory is automatically mapped to a particular address range by the underlying architecture, an address within that range can also be used to obtain access to the memory so it can be used as the backing store.

to access it, then it will get an exception thrown. Note that the above class has full control over the interface it provides. Some methods available on scoped memory are not appropriate in this context (e.g. joinAndEnter) and therefore are not exposed in the interface.

Unfortunately, this approach does not fully encapsulate the class. The memory area can escape. In the RTSJ, a real-time thread can ask the memory area of an object to be returned using the static getMemoryArea method in the MemoryArea class. If a real-time thread does this, and saves the reference in a static field then another thread can access[3] the memory area directly.

An alternative approach is to subclass the LTPhysicalMemory class and provide the explicit checks for the owner of the scope.

```
public class PrivateScope extends LTPhysicalMemory {
  public PrivateScope(long size) {
    super(PhysicalMemoryManager.ON_CHIP, size);
    this.owner = RealtimeThread.currentRealtimeThread();
  }

  public void enter(Runnable R) {
    if (RealtimeThread.currentRealtimeThread() != owner)
    {
      // throw an exception, the calling thread
      // does not own this scope
    }
    else super.enter(R);
  }

  // similarly for executeInArea, newArray and
  // newInstance etc

  private RealtimeThread owner;
  . . .
}
```

This is secure, but has the disadvantage that all the functionality of scoped memory not needed in private memory has to have its associated methods overridden.

Of course the two approaches can be merged, as shown below.

```
public class ThreadLocalScope {
   public ThreadLocalScope(long size) {
     myMemory = new ThreadLocal<PrivateScope>();
     PrivateScope myMem = new PrivateScope(size);
       // can throw out of memory
     myMemory.set(myMem);
   }

   public void enter(Runnable R) {
     // unchanged
   }

   public void executeInArea(Runnable R) {
     // unchanged
   }

   // other needed scoped memory methods

   private ThreadLocal<PrivateScope> myMemory;
}
```

Now the programmer can be given the appropriate interface, but the approach is still secure.

3.2 Pinable Memory Areas

Version 1.1 of the RTSJ will provide the notion of a pinable scoped memory area. This is a memory area that can be kept open even when there are no threads currently entered. In the context of this paper, this is a very useful extension, as it allows the thread to leave the local scoped memory area and enter into another allocation context. It can then return and continue to access any objects created.

3.3 Garbage Collection

Object stored in a processor's private local memory does have implications for any garbage collection being performed. Whilst the memory assignment rules of the RTSJ will ensure that objects in heap and immortal memory cannot reference any processor-local memory, references from local scopes to the heap are allowed. Hence, the processor-local memory must be scanned by the garbage collector. Consequently, parallel collection must be performed where each processor has its own thread which can scan its local memory area. Another possibility is to extend the idea of non-blocking root scanning [11]. To decrease the blocking time during root scanning the thread local root scanning is delegated to the mutator threads. If requested from the collector thread, each thread scans its own local root set at the end of its period. This root scanning phase can be extended to include the local memory area scanning.

4 Evaluation

For an evaluation of the concept we implemented the local memory in a chip-multiprocessor version of the Java processor JOP [14, 8]. An on-chip memory of 1 KB is attached locally to each processor. It has to be noted that JOP does not support the full RTSJ, but is intended as a real-time Java platform for the safety-critical Java subset. Therefore, we evaluate the scope cache within this context.

Figure 2 shows an example of the JOP CMP system. Each processor core contains its private method cache (M$), stack cache (M$), and the scratchpad memory (SPM). The cores are connected via an arbiter to the external, shared main memory.

[3] We are grateful for Fridtjof Siebert for pointing out this flaw in our original proposal.

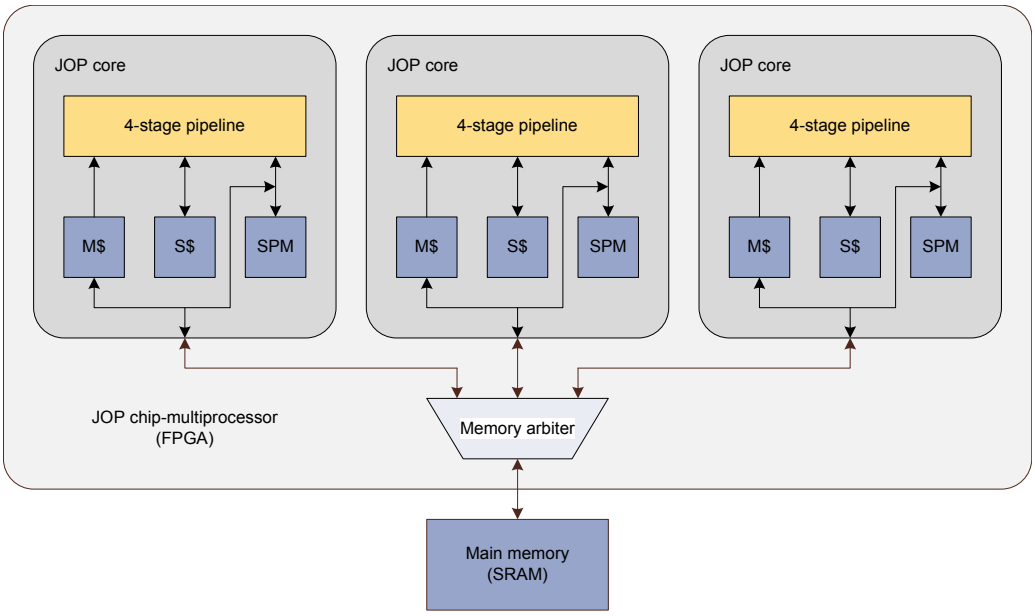

Figure 2. A three core JOP CMP system with three core-local memory areas

4.1 JOP Caches

Two time-predictable caches are proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

As the stack is a heavily accessed memory region, the stack – or part of it – is placed in on-chip memory. This part of the stack is referred to as the stack cache and described in [13]. The stack cache is organized in two levels: the two top elements are implemented as registers, the lower level as a large on-chip memory. Fill and spill between these two levels is done in hardware. Fill and spill between the on-chip memory and the main memory is subjected to microcode control and therefore time-predictable. The exchange of the on-chip stack cache with the main memory can be either done on method invocation and return or on a thread switch.

In [12], a novel way to organize an instruction cache, as method cache, is given. The idea is to cache complete methods. A cache fill from main memory is only performed on a miss on method invocation or return. Therefore, all other bytecodes have a guaranteed cache hit. That means no instruction can stall the pipeline.

The cache is organized in blocks, similar to cache lines. However, the cached method has to span continuous[4] blocks. The method cache can hold more than one method. Cache block replacement depends on the call tree, instead of instruction addresses. This method cache is easier to analyze with respect to worst-case behavior and still provides substantial performance gain when compared against a solution without an instruction cache. The average case performance of this method cache is similar to a direct mapped cache [12]. The maximum method size is restricted by the size of the method cache. The pre-link tool verifies that the size restriction is fulfilled by the application.

Both caches do not need a cache coherency protocol in the case of multiprocessing: the stack cache contains only thread local data and the method cache contains just instructions that are not updated by a program.

4.2 Scope Cache

JOP does not contain a data cache and therefore also avoids the bottleneck of the cache coherence protocol in a CMP system. However, when all access to heap allocated data goes directly to the main memory, the memory bandwidth becomes the limiting factor for a CMP system. Therefore, we add a program managed local memory, the proposed local scope cache, to each processor core. The memory is mapped to a well known address and the boot process detects the size automatically. The memory is used as backing store for the scoped memory.

4.3 WCET Analysis

JOP is designed as a time-predictable processor to simplify the WCET analysis. Most bytecode instructions are executed in a constant number of cycles. The execution time of bytecodes that access memory (e.g., field and array access) depends on the memory timing. In the case of a CMP system

[4]The cache addresses wrap around at the end of the on-chip memory. Therefore, a method is also considered continuous when it spans from the last to the first block.

the execution time also depends on the memory arbitration policy and the number of cores in the system. A time division multiple access (TDMA) based arbiter allows to bound the WCET to a reasonable value [7]. The fair arbiter, which is also evaluated in the following section, performs better in the average case, but the WCET for a memory access is high. As the loading of the method cache cannot be interrupted in that arbiter, the WCET of a single memory access includes the cache load of all other cores.

The access time to the on-chip memory is a single cycle. Therefore, all memory access to objects that are allocated in the local scope cache have a short and constant execution time. Integration into the WCET analysis tool can be performed by additional annotations of the source code. A better approach would be to detect the different memory types and which objects are allocated in them by data-flow analysis.

4.4 Measurements

We have built different configurations of the CMP system with 1, 2, 4, and 8 processor cores. As an evaluation platform we use the Altera DE2 board with a Cyclone-II FPGA. All designs are clocked at 90 MHz and the main memory is a 16-bit SRAM with an access time of 4 cycles for a 32-bit read operation and 6 cycles for a 32-bit write operation. All configurations consume the same amount of on-chip memory per core: 1 KB stack cache, 1 KB method cache, and 1 KB local scope cache. The size of the method cache is chosen smaller than usual, but only with this configuration we are able to synthesize a 8 core version of the CMP system.

As an arbiter we used two different types: the *fair* arbiter [8] and the *time-triggered* (TDMA) arbiter [7]. Both arbiters distribute the bandwidth equally to all cores. However, the fair arbiter performs dynamic arbitration each cycle and results in a better average case performance of the system than the TDMA arbiter. The TDMA arbiter divides the access time into equal time slots. The size of the time slot can be configured. We choose the shortest possible size of 6 cycles. With the TDMA arbiter the WCET of Java bytecodes can be determined easily, whereas the dynamics of the fair based arbiter complicates the low-level WCET analysis.

For the evaluation we use a benchmark that performs multiplication of two 100x100 matrices. The work is distributed at the basis of rows that are calculated. Each core tries to get a new row for the calculation when finished the former one. The local scope cache is used to cache one row during calculation. In the inner loop of the multiplication one memory access is performed to the main memory and one memory access to the local scope cache. We compare this calculation with a version without caching where two main memory accesses are performed in the inner loop.

Table 1 shows the execution time of different configurations of the CMP system with and without local scope caching. The execution time is shorter than reported in [8] as we have optimized the benchmark code. For this kind of processing task we assume that the code is optimized to avoid memory access as much as possible in a CMP system.

Arbiter	Execution time			
	Fair		TDMA	
# Cores	w/o cache	w cache	w/o cache	w cache
1	839 ms	761 ms	839 ms	761 ms
2	473 ms	383 ms	625 ms	489 ms
4	315 ms	202 ms	488 ms	353 ms
8	307 ms	161 ms	389 ms	260 ms

Table 1. Execution time of the matrix multiplication in ms for different CMP and arbiter configurations

From the table we see several trends: (1) the fair arbiter delivers a better average case performance than the TDMA arbiter; (2) this type of processing scales without caching only for two cores; (3) the local caching of one data row provides some performance increase for a single core; and (4) local caching helps to scale up to 4 or 8 cores.

Figure 3 shows the performance relative to a single core system without scope caching with the fair arbiter.[5] We can see that the system saturates the memory bandwidth with 4 cores without caching. With scope caching we can still achieve a performance improvement of a factor of 5 with cores. We are quite surprised by this improvement as we do not expect a linear performance increase with the number of cores. As the numbers of transistors increases exponentially we consider a logarithmic performance increase with the number of processors a success. However, the chosen benchmark is trivial to parallelize. The result shows that DSP like algorithms can benefit from the local scope cache. The improvement of more than 2 and 4 for 2 and 4 cores respectively results from the comparison against the non-caching version.

The same data is shown in Figure 4 with the TDMA arbiter. The time-predictable nature of this configuration results in a lower average-case performance. This is expected. The CMP scales with the number of processors, but slower as with the fair arbiter. In this case the performance increase is around 35% for the non-cached solution and around 70% for the cached solution when the number of processors is doubled. A logarithmic improvement that we consider practical.

4.5 Discussion

In Section 3.1 it is suggested that the implementation of LTPhysicalMemory communicates with the PhysicalMemory-

[5] Execution time of the single core system divided by the execution time of the CMP system

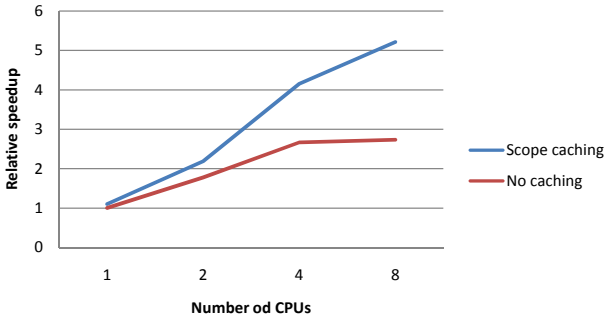

Figure 3. Performance of different CMP systems with the fair arbiter

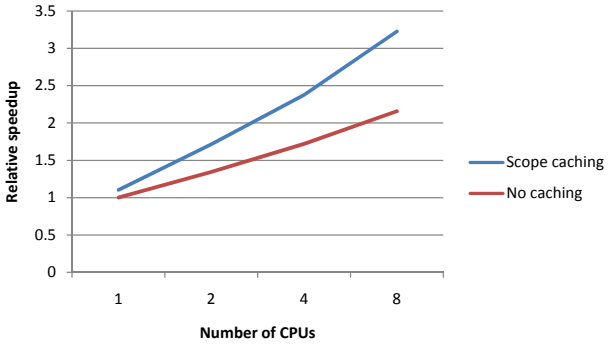

Figure 4. Performance of different CMP systems with the TDMA arbiter

Manager to request a scratchpad memory area that serves as backing store for the thread local scope. The actual implementation is JVM and operating system specific. A scratchpad memory is usually mapped to a well known address. An implementation of the JVM can also directly use that knowledge to allocate the scope backing store at that specific address. For example, a scratchpad memory of 1 KB at address 0x1000000 can be requested with following constructor:

```
LTPhysicalMemory(PhysicalMemoryManaged.ON_CHIP_PRIVATE,
  0x1000000, 1024)
```

In the implementation on JOP normal, scoped memory is implemented with the help of plain Java arrays. The array is the backing store for the scope and is allocated in immortal memory. In the implementation of bytecode new a thread local data structure is examined to distinguish between the different memory areas. To request the on-chip memory as backing store we use the convenient abstraction of hardware objects [15]. A hardware object is either a Java object or a Java array that is mapped to a distinct memory area where an I/O device is located. The hardware objects and arrays provide a Java based lightweight abstraction of I/O devices. For the thread-local scope cache an array, mapped to the on-chip memory address, is requested from a system internal factory. No further distinction between an array that represents the backing store for a normal scope or a PrivateScope is necessary in the implementation. The constant ON_CHIP_PRIVATE is used to distinguish between the two different types within the constructor. In the actual implementation this constant is passed to a package private constructor of ScopedMemory where the allocation of the backing store is performed.

In this paper on-chip memory is tied to a single thread that is not allowed to migrate between different cores. If the thread is allowed to migrate, the on-chip memory content also needs to migrate to a different core. Technically migration of the on-chip memory content is possible, but the scheduling cost is increased accordingly. In our opinion thread migration is too costly and too hard to analyze. However, sharing of a scratchpad memory between threads that are pinned to the same core is a valuable option. For this extension the runtime check on enter() needs to be changed to check if the thread is running on that core where the PrivateScope was allocated.

Thread local data is per definition not shared. Therefore, the expensive cache coherence protocol can be avoided for local scope caching. Cache coherence protocols do not scale well and also introduce hard to analyze latencies for the memory access.

5 Related Work

A common solution to avoid data caches is an on-chip memory, named scratchpad memory, that is under program control. This program managed memory implies a more complicated programming model. However, scratchpad memory can be automatically partitioned [2, 1, 18]. A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [9] and the allocation of data in the scratchpad memory [19, 17, 4] can be optimized for the WCET. A comparison between locked cache blocks and a scratchpad memory with respect to the WCET can be found in [10]. While former approaches rely on the compiler to allocated the data or instructions in the scratchpad memory an algorithm for runtime allocation is proposed in [6].

Exposing the scratchpad memory at the language level, as proposed in this paper, can further help to optimize the time-critical path of the application.

With a core local scratchpad memory the data allocated in it cannot be shared between threads on different cores. Program analysis to detect thread-local heap data is presented in [16]. This analysis can help to automatically allocate data in the scratchpad memory on a CMP system.

6 Conclusion

The search for increasingly fast processors is leading towards CMP solutions. These will inevitably have both processor-local and shared memory modules. In this paper we have shown that the memory management framework provided by the RTSJ is flexibly enough to allow processor-local memory to be effectively accessed by real-time threads through the notion of thread-local scoped memory areas. The evaluation shows that significant speedup can be obtain on highly parallel data-oriented algorithms where data can be cached locally.

Throughout this work, we have assumed that threads that access thread-local scopes do not migrate between processors, other threads may. Hence, we are trading off the timeliness of memory accesses, the simplicity of the hardware architecture and the predictability of the resulting schedulability analysis against average case benefits and any theoretical schedulability improvements that might be obtained by allowing a thread migration.

Acknowledgement

This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD). The authors would like to acknowledge the contribution made by the partners to this work, in particular Fridtjof Siebert.

References

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*, pages 318–326, New York, Oct. 30 Nov. 01 2003. ACM Press.

[2] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[3] M. Barr. Memory types. *Embedded Systems Programming*, 14(5):103–104, May 2001.

[4] J.-F. Deverge and I. Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[5] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[6] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM.

[7] C. Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[8] C. Pitter and M. Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[9] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[10] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2007)*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[11] W. Puffitsch and M. Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.

[12] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[13] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[14] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[15] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[16] B. Steensgaard. Thread-specific heaps for multi-threaded programs. *SIGPLAN Not.*, 36(1):18–24, 2001.

[17] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 223–232. IEEE Computer Society, 2005.

[18] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst*, 14(8):802–815, 2006.

[19] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE2005).*, pages 600–605 Vol. 1, March 2005.

[20] A. Wellings. Multiprocessors and the real-time specification for java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.