Portable Memory Consistency for Software Managed Distributed Memory in Many-Core SoC

Jochem H. Rutgers, Marco J.G. Bekooij, Gerard J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
j.h.rutgers@utwente.nl

Abstract—Porting software to different platforms can require modifications of the application. One of the issues is that the targeted hardware supports another memory consistency model. As a consequence, the completion order of reads and writes in a multi-threaded application can change, which may result in improper synchronization. For example, a processor with out-of-order execution could break synchronization if proper fence instructions are missing. Such a bug can cause sporadic errors, which are hard to debug.

This paper presents an approach that makes applications independent of the memory model of the hardware, hence they can be compiled to hardware with any memory architecture. The key is having a memory model that only guarantees the most fundamental orderings of reads and writes, and annotations to specify additional ordering constraints. As a result, tooling can transparently and properly implement fences, cache flushes, etc. when appropriate, without losing flexibility of the hardware design. In a case study, several SPLASH-2 applications are run on a 32-core software cache coherent MicroBlaze system in FPGA. Moreover, this approach also allows mapping to scratch-pad memories and a distributed shared memory architecture.

I. INTRODUCTION

With the growth in the number of mobile and embedded devices, porting software to various platforms is becoming increasingly important. Programmers not only face different software contexts (OSs and APIs), but also different hardware architectures with various numbers of cores and communication infrastructures. Porting to other hardware often requires subtle, but fundamental changes to the software, due to a changed memory consistency model, which defines the order in which writes are observed by the processors in the system.

The ‘natural view’ on memory is defined by the Sequential Consistency (SC) [1] model, that more or less defines that all processors see changes to the memory exactly the same. Such a strict model simplifies programming, but it is hard to implement it efficiently in hardware, because of globally atomic constraints, like that a write should be visible for all processors at the same time. For sake of performance and scalability, processor and system designers weaken the ordering constraints. For example, processors with out-of-order execution can reorder two writes, but this order can be important for synchronization.

If an application is designed for hardware with one specific memory model, there is no guarantee that it will work

correctly and efficiently on other hardware. Even if the application initially seems to work, sporadic races can still occur and are hard to debug.

This paper presents Portable Memory Consistency (PMC), that defines a memory model (referred to as the *PMC model*) and an approach to apply this model to an application and any memory architecture by means of annotations to the source code (the *PMC approach*). Traditionally, a memory model is seen as a contract between hardware and software and defines the semantics of reads and writes. In contrast, we use our memory model as an abstraction layer that disconnects the application from the underlying hardware. The key is that all orderings that are required by the application, are made explicit. Then, tooling can fill in the gap between what the application requires and which orderings are satisfied by the hardware. As a result, porting applications to hardware with another memory model becomes just a compiler setting.

For this, we propose a single, weak, synchronized memory (consistency) model that only defines five memory operations and four types of orderings between them. This model 1) is strong enough to mimic SC when required by the application; 2) is weaker than Entry Consistency [2], because synchronization operations to different memory locations are unordered, unless explicitly specified by fences; and 3) allows mapping to all existing hardware, because it is an intersection of all common memory models. Since changing a memory model of an existing programming language is impossible—we use C and C++ in our experiments—it is required that the source code is annotated to indicate which orderings are required by the application¹.

The PMC approach involves that an application is designed and annotated for the PMC model, regardless of the targeted hardware. The PMC model is designed such that a mapping of the primitives and ordering relations to specific hardware can be designed and verified with relative ease. So, because all required orderings are made explicit, the compiler c.q. platform can use this information to take all measures in either software or hardware to ensure the orderings and synchronization on the hardware at hand, without losing flexibility of optimization of other non-ordered operations.

¹ Although using the PMC model natively in the semantics of a new programming language is the best way to go, this is beyond the scope of this paper.

The structure of this paper is as follows. First, related work is discussed in Section II. The basic idea behind PMC is presented in Section III. Our solution consists of a memory model (Section IV) and annotations, which result in an abstraction from the underlying hardware. This allows compiling applications to completely different memory architectures, of which three are discussed in Section V: software cache coherency, a distributed shared memory architecture and one with scratch-pad memories. As as proof of concept, three applications of the SPLASH-2 benchmark set have been annotated and implemented on a 32-core software cache coherent MicroBlaze system in FPGA. Section VI presents the results, and discusses additional example applications that are mapped to the two other architectures. Section VII concludes the paper.

II. RELATED WORK

In the past few decades, a lot of work has been done on memory models. The main motivation for defining different weakness in memory models is to achieve efficiency of the hardware implementation. Nevertheless, these models have a strong mathematical basis. Most work focuses on the model itself and, to the best of our knowledge, no work directly relates such formalism to how it is implemented in hardware and used by applications in practice. For example, memory models require that the source code is properly labeled c.q. annotated [3], but do not discuss in detail how the annotation should be used. This paper links the models to annotations in the source code and to the implementation on concrete hardware.

Memory models can be grouped in two classes: uniform and synchronized. Uniform models have only two operations on the memory: read and write. Important uniform models include Sequential Consistency [1], where all operations are in total order; Processor Consistency (PC) [4, 5], where different processors can disagree on the observed order of operations to different locations by different processors; and Slow Consistency [6], where only operations of one processor to the same location is guaranteed.

Synchronized models define special operations, usually acquire and release. These guarantee mutual exclusive access to specific memory locations. Ordinary reads and writes are usually ordered like Slow Consistency, but the orderings between acquires and releases differ. Among others, Release Consistency (RC) [3] and Entry Consistency (EC) [2] order acquires and releases like PC, but are different at which acquire/release pairs on different locations are allowed concurrently. Our PMC model is even weaker than EC, but our approach includes specifying annotations, such that PC-equivalent strictness can be achieved.

Other weaker models than PC and EC do exist, but their usability is limited. For example, GS-Location Consistency [7] is one of the weakest synchronized models, but Long et al. [8] point out that specific algorithms cannot be implemented.

Initially: `flag=0`

Process 1:

```
1 X = 42;
2 flag = 1;
```

Process 2:

```
3 while (flag!=1)
4     sleep();
5 print (X);
```

Figure 1. A Sequentially Consistent correct program, which breaks on an architecture with two memories

Moreover, Frigo [9] states that any implementation will result in a stronger model. Furthermore, PRAM [10] is weaker than PC, but because certain nondeterminism is allowed, programming for it is hard [9].

Steinke and Nutt [11] analyze memory models, and give a taxonomy that is based on the models’ common properties. They discuss 13 uniform models (and conclude that there can be more) and define synchronized models as combinations of the uniform ones. Their discussion focuses on formal properties, which do not (easily) allow an implementation. In contrast, we describe a concrete implementation of the memory model we propose in this paper.

Integration of a memory model in a programming language is preferable, such that tooling can verify or complement ordering constraints. The latest C++ standard (C++11 [12]) includes multithreading and defines a memory model. It assumes that the programmer can identify variables that should be declared atomic and access it accordingly. However, Batty et al. [13] conclude that this model is not clearly defined by the standard and the corresponding mathematical model might not be ‘sufficiently widely accessible’. In this paper, we define a model that is kept minimalistic, which simplifies reasoning about behavior.

III. THE PROBLEM WITH MEMORIES

Porting software to hardware with another memory model can cause very subtle problems. Fig. 1 shows an example of this. The program of the figure intends to communicate the value 42 from process 1 to 2 via variable `X`. On a platform that implements SC, this program will behave correctly.

However, the program will break when it is ported to a hardware architecture that is also depicted in Fig. 1. The essence of the problem is that the latency of the write operation by process 1 to the memory that holds `X`, is higher than that of `flag`. When process 2 polls the `flag`, it first reads 1 and then reads `X`. Because of the high latency of the write of `X`, process 2 can read the old value of `X` before 42 has arrived in the memory—the program breaks. Tracking down this bug is non-trivial by looking at the source code and could even be more difficult to find when the latencies in the interconnect vary over time. The problem cannot be

prevented, even if both `X` and `flag` are declared **volatile**, atomic or separated by fence instructions.

The underlying problem in this architecture is that the order of the two writes of process 1 is not guaranteed, as is the case for SC. The behavior of the memory—which is distributed in this example—is defined by a memory (consistency) model, which defines the conclusions a *process* can draw when it *observes* state changes of *locations* of the memory and whether different processes c.q. observers must *agree* on these conclusions. In the example, the conclusion that every process agrees that 42 is visible before the `flag` is set, is wrong, even though the write of `X` is initiated first.

The basic idea of our approach is that there are as few assumptions of ordering of operations in the source code as possible and that all additional constraints should be defined explicitly. So, the solution is two-fold: a weak memory model, and annotations for additional constraints. This memory model, which will be discussed in more detail in Section IV, can be summarized as that it is only guaranteed that reads and writes from the same process(or) to the same location will be observed in the same order. Additionally, the annotations allow a compiler to insert special memory operations, which enforce an order between two operations of different locations by one process (a proper fence), and two operations on the same location by multiple processes (acquire/release).

If the source code indicates that the write to `X` and `flag` should be observed in that specific order, then a compiler or OS can enforce it. For example, a compiler can insert a read of `X` between the writes to `X` and `flag`. Because the read will be completed after `X` has been written, it is guaranteed that every other processor will first observe the change to `X` and then `flag`.

A consequence of disconnecting the memory model of the hardware and the one an application is designed for is that the strictness of the hardware becomes just a feature. This is similar to having hardware floating point support in a processor: a programmer can always use floating point operations in an application, but it is more efficient when the hardware supports it, otherwise software emulation is used. Similar, synchronization can always be used, but when the memory model of the hardware is stricter, the application can be more efficient.

We discuss the memory model, annotations and implementations in more detail in the next three sections.

IV. MEMORY MODEL

This section proposes a weak, synchronized memory model. This model is the programmer's view on memory in the PMC approach.

A. Base Model

A program defines a partial order of operations on memory locations, which can be represented as a directed, acyclic dependency graph. In general, different concurrent processes can observe operations in different order. However, the edges

Table I
ORDERINGS BETWEEN EXISTING AND NEW OPERATIONS ON LOCATION v BY PROCESS p

		pattern	new operation				
			r	w	R	A	F
operation	read	$(r, p, v, *)$	$\prec_\ell$	$\prec_\ell$	$\prec_\ell$		$\prec_\ell$
	write	$(w, p, v, *)$	$\prec_\ell$	$\prec_P$	$\prec_P$		$\prec_\ell$
	acquire	$(A, p, v, *)$	$\prec_\ell$	$\prec_P$	$\prec_P$		$\prec_F$
	release	$(R, p, v, *)$				$\prec_S^\dagger$	$\prec_F$
	fence	$(F, p, *, *)$			$\prec_F$	$\prec_F$	$\prec_F$

[†] An acquire has its ordering $\prec_S$ on $(R, *, v, *)$, not just on releases of the same process.

in the graph indicate which operations are ordered in time, independently of who observes it. These dependencies can partly be analyzed at compile-time, but some parts are only known at run-time, due to data dependencies, for example. In run-time, all dependencies are known—although such graph is never actually stored. Such state in run-time is an *execution* of a program. For the base model, we use a notation that is similar to the one as proposed by Steinke and Nutt [11].

Definition 1 (Execution). *An execution is a model of the state of a program at one moment in time and is defined as $E = (P, V, O, \prec)$, where*

- P is the set of all processes.
- V is the set of all shared variables, c.q. locations.
- O is the set of all issued operations.
- The transitive, binary relation $\prec$ is a partial order on O .

Among other details that will be explained further on, Table I lists all operations and their abbreviations. Reads and writes of a memory location in V should be atomic. In general, only bytes are indivisible. Handling variables that span multiple bytes is covered in Section V. The table also lists patterns to match operations.

Definition 2 (Pattern). *The pattern (operation, p, v , value) is a subset of O where $p \in P$ and $v \in V$, which matches any $o \in O$ that have the same properties. A $*$ matches all.*

So, the pattern $(w, *, v, *)$ matches all writes to location v by any process, for example. Next, the initial state of a program is defined as:

Definition 3 (Initialization). *An execution $E = (P, V, O, \prec)$ is initialized, such that P contains all processes, V contains all locations, and $\prec$ is empty. All locations have an initial operation that behaves like a write and release, so O is initialized, such that $\forall v \in V : |(\{w, R\}, \epsilon, v, \perp)| = 1$, where ϵ is equivalent to all processes.*

Definition 3 states that all locations have an initial operation that is both a write and release. As a result, reads and acquires always have a predecessor.

Process 1:

```
1 X = 1;
2 X = 2;
```

Figure 2. Program order of two writes

Process 1:

```
1 X = 1;
2 if (X==1)
3   X = 2;
```

Figure 3. Local order of a read

B. Operations

A program issues operations to the memory system. All operations that can be executed by any process are:

- *read*: retrieves the value of a previously executed write operation of a specific location.
- *write*: replaces the value of a location. Writes do not have to be visible for all processes immediately.
- *acquire*: gets an exclusive lock on a specific location. An acquire must be followed by a release of the same process. Moreover, mutual exclusion between an acquire and release must be guaranteed.
- *release*: gives up the exclusive lock on a specific location.
- *fence*: adds dependencies to locally executed operations.

The properties of the operations are more formally discussed in Section IV-D. When operations are executed, they add orderings to the execution graph that is being constructed.

Definition 4 (State transition). *When an operation o on location $v \in V$ by process $p \in P$ is executed, the next execution is $E' = (P, V, O', \prec')$, where $O' = O \cup \{o\}$, and $\prec'$ extends $\prec$ such that the ordering rules as indicated in Table I apply to all matching operation patterns and o .*

Without explaining those ‘ordering rules’ at this point, Table I defines the rules that are applied between operations. For example, when a new write operation is executed, it will add the orderings $\prec_\ell$ between all previously executed reads on the same location by the same processor and the new write, and it will similarly add the orderings $\prec_P$ between all previous writes and acquires and the new write. So, the dependency graph grows by every new operation and these orderings are never removed. The next subsection discusses these different types of orderings in the table in more detail.

C. Orderings

Fig. 2 shows a simple program by one process that executes two writes to the same location X . The graph shows that when $X=1$ is executed, one dependency is added from the initial write. This is graphically presented as $(A) \xrightarrow{\prec_*} (B)$, which indicates that every process observes that A occurred before B because of the indicated ordering rule, where $\prec_*$ stands for any rule. When $X=2$ is executed, a dependency is added from all previous writes to the new one. We omit the (implicit) initial write in the figures. Moreover, the figures are transitively reduced; all redundant orderings are left out of the figures, like the one from the initial write to $X=2$. The rule in this example is the *program order*.

Definition 5 (Program order). *Program orderings $\prec_P$ are globally visible orderings between two operations of one process on one location.*

Definition 5 implies that writes of one process to different locations can be observed in a different order by different observers. Every process observes writes to the same location by one process in the same order, but the effect of the write does not have to be instantaneously visible.

A read will add ordering constraints that are only visible to the *local* c.q. executing process. Fig. 3 gives an example. In this case, there is a relation between $X=1$ and the consecutive read; the compiler or hardware should not reorder these two operations. As a result, the read can only return the value 1.

Definition 6 (Local order). *Locally visible orderings $\prec_\ell$ are only visible to the executing process p .*

Graphically, a local ordering is denoted $(A) \xrightarrow{\prec_\ell} (B)$, where only the executing process observes A occurring before B . All other processes could disagree. With this order, all local control dependencies in the program are preserved. The reads, writes, local and program order as discussed so far are equivalent to Slow Consistency.

Because the program order $\prec_P$ only orders per process, operations of two processes accessing the same location can be interleaved in any way. For inter-process orderings, synchronization is added. Synchronization consists of two operations: *acquire* and *release*, which behave in a normal fashioned mutual exclusive way.

Definition 7 (Synchronization order). *Synchronization orderings $\prec_S$ are globally visible, per location orderings that can span multiple processes.*

Fig. 4 shows a program with two processes that both try to acquire the same location. Depending on which process will get the lock first, process 1 reads either 0 or 2 (the latter is depicted in the figure). The figure shows how different ordering rules of Table I are applied.

Until now, it is impossible to enforce orderings between two locations. However, a communication pattern like in Fig. 5 is very common, where data X is communicated by setting a flag and another process waits until it receives the flag before reading the data. For that, a *fence* is needed.

Definition 8 (Fence order). *Fence orderings $\prec_F$ are globally visible, per process orderings that can span multiple locations.*

Initially: $X=0$

Process 1:

```
1 acquire(X);
2 r = X;
3 release(X);
```

Process 2:

```
4 acquire(X);
5 X = 1;
6 X = 2;
7 release(X);
```

Figure 4. Exclusive access with two processes with a dependency graph of one possible interleaving. Regardless of which interleaving happens in run-time, every observer agrees on that interleaving.

Initially: $f=0$

Process 1:

```
1 acquire(X);
2 X = 42;
3 fence();
4 release(X);
5
6 acquire(f);
7 f = 1;
8 release(f);
```

Process 2:

```
9 while (f!=1)
10   sleep();
11 fence();
12
13 acquire(X);
14 r = X;
15 release(X);
```

Figure 5. Simple multi-core communication example

The fence of line 11 prevents the compiler from moving the acquire at line 13 to before the while loop, where it (potentially) can acquire the lock before X is written. The dotted arrow indicates that when f is eventually observed being 1, it can be concluded that write of 1 must have been executed before. Although none of the ordering rules enforce it, this control dependency is valid, but only locally known to process 2. When process 2 acquires X afterwards, the fences make sure that it will always acquire after process 1 has acquired (and released) it. Therefore, it is guaranteed that process 2 will read the value 42. Note that there is no way for process 2 to make sure the value 42 of X is read at line 14, without acquiring it; then there is no chain of dependencies that lead to the write of 42.

Finally:

Definition 9 (Global order). *The set of globally visible orderings $\prec_G := \prec_P \cup \prec_S \cup \prec_F$ are orderings that all processes always agree on, no matter how the effects of the orderings are observed.*

Definition 10 (Execution order). *The set of execution orderings $\prec := \prec_G \cup \prec_\ell$ is a partial order on all operations of an execution.*

Because now processes can have a different views on the orderings, the point of view is included in the ordering relation. For two operations $a, c \in O$, we use the shorthand notation $a \prec c$ for describing $a \prec_G c$ —the local orderings are not included, as the notation does not indicate the point of view. Additionally, $a \stackrel{p}{\prec} c$ includes both the global ordering and the local orderings of process p . So, this can recursively be described as: $a \stackrel{p}{\prec} c$ iff $\exists b \in O : a \prec_G b \stackrel{p}{\prec} c \vee a \stackrel{p}{\prec}_\ell b \stackrel{p}{\prec} c$.

D. Observing Slowly

Based on the ordering rules above, various properties of the operations can be defined more precisely. The last write operation of a location is the one that you first encounter when following the dependency graph in reversed direction.

Definition 11 (Last write). *The last write to $v \in V$ before operation $o \in (*, *, v, *)$ is denoted $W_o = \{a \in (w, *, v, *) | a \prec o \wedge \nexists b \in (w, *, v, *) : a \prec b \prec o\}$.*

W cannot be empty, because at least the initial write is included. If W contains multiple writes, reading the location is nondeterministic; a data-race occurred. This leads to the conclusion that for a deterministic application, all writes to a single location must be in total order. As Table I shows, ordering between writes to the same location of two processes is only possible via acquires and releases. Therefore, all writes must be enclosed by an acquire and release—but a single acquire/release pair can contain multiple writes.

Definition 12 (Read value). *A read operation o by process p from location v returns either the last written value according to observed dependencies, or any value that is written afterwards. So, o can read $\{\text{value}(b) | a \in W_o, b \in (w, *, v, *) : a \stackrel{p}{\prec} b\}$. However, when two read operations $o \stackrel{p}{\prec} o'$ read from the write operations w and w' , respectively, then this implies $w \stackrel{p}{\prec} w'$.*

So, a read can return an already overwritten value, because writes slowly propagate through the system. However, it is impossible to return an older value when previously a newer value has been returned. A formal description of such observer function is given by Frigo [9].

In Fig. 5, process 2 polls the flag. However, there is no control over when the write of process 1 arrives at process 2. It makes sense that a platform provides a *flush* function that makes writes globally visible sooner, but because the flush cannot be used to guarantee ordering, this is more a convenience; it is not part of the memory model.

Moreover, the fences discussed in this section are applied on all locations. Without loss of generality, one could offer more complex fences on specific locations for optimization purposes, but this is beyond the scope of this paper.

E. Comparison to Existing Models

As stated above, the orderings and behavior of the read and write operations of PMC is identical to Slow Consistency. The globally observable orderings $\prec_G$ can also be described by two properties: 1) $\prec_P \cup \prec_S$ results in an order per location that spans multiple processes, which is equivalent to Global Data Order (GDO), as defined by Steinke and Nutt [11]; 2) $\prec_F$ is a order per process that spans multiple locations, which is equivalent to Global Process Order (GPO) [11]. For most synchronized relaxed models, Slow Consistency is assumed for reads and writes, and then different flavors of synchronization are added.

When the writes to shared variables are wrapped in an acquire/release pair—which is necessary in order to be data-race free—the writes to a single location are in total order. As a result, the behavior is identical to Cache Consistency (CC); total order of writes per location and ‘slow reads’, where values propagate slowly through the systems. However, just having CC is not enough to implement the communication in Fig. 5; fences are required. If one would add a fence between every operation, the model is equivalent to Processor Consistency (PC); total order of all writes per location (GDO) and total order of all writes per process (GPO).

We argue that it is required that the platform supports both GPO (c.q. fences) and GDO (c.q. acquire/release pairs). Without GDO, which is the case for PRAM, nondeterministic execution cannot be confined and writing applications becomes extremely hard [9]. However, without GPO, it is not possible to simulate Sequential Consistency (SC) [14]. Relaxing the total order requirement of GDO to a partial order is proposed by Gao and Sarkar [7], but any implementation of it will be stronger [9]. So, both GDO and GPO are required to be usable, which is precisely what our model is based on.

Because it is possible in our model to apply all ordering constraints required to behave like PC, our model can benefit of all properties of PC, such as that is able to simulate SC for data-race free programs [4]. However, our models allows specifying only the essential orderings, where PC overly constrains the possible orderings.

Compared to EC, our model is weaker, because of two additional relaxations: 1) exclusive access (between acquire/release) is allowed alongside read-only access; and 2) acquire/releases of different locations by the same process are not ordered, unless a fence is applied.

V. ANNOTATION AND ABSTRACTION

Ideally, the PMC memory model as discussed in Section IV should be the native model of a programming language and the semantics of that language should only define orderings of the model. In that case, programmers specify all required orderings in an intuitive way. For now, such language does not exist, so we introduce annotations that can be used in (existing) C programs. Adding ordering information by means of annotations is essential in the PMC approach.

Initially: $f=0$

Process 1:	Process 2:
1 <i>entry_x</i> (X);	10 do { <i>entry_ro</i> (f);
2 X = 42;	11 poll = f;
3 <i>fence</i> ();	12 <i>exit_ro</i> (f);
4 <i>exit_x</i> (X);	13 } while (poll!=1);
5	14 <i>fence</i> ();
6 <i>entry_x</i> (f);	15
7 f = 1;	16 <i>entry_x</i> (X);
8 <i>flush</i> (f);	17 r = X;
9 <i>exit_x</i> (f);	18 <i>exit_x</i> (X);

Figure 6. Properly annotated source code of Fig. 5

A. Front-end: Annotations in Source Code

Accesses to non-shared objects do not have to be annotated. As stated before, all writes to shared objects should be wrapped in an acquire/release pair. For symmetry reasons, *all* reads and writes should be wrapped, in either an entry/exit pair with exclusive read/write access (like acquire/release) or non-exclusive read-only access. Together with reads and writes, the annotations below covers all operations of Table I.

- *entry_x*(X): Issues an acquire operation on X. An *entry_x*() should be paired with an *exit_x*() .
- *exit_x*(X): Issues a release operation on X. During an *exit_x*(), all writes to X do not necessarily have to be notified to others. An implementation could do a ‘lazy release’, which keeps all modifications to X local, until another process does an acquire of X. An eager release implementation would do a *flush*(X) (see below) before giving up the lock on X.
- *entry_ro*(X): Marks the start of non-exclusive, read-only access to X. In the implementation of this call, the system could do some effort to retrieve updates of X. An *entry_ro*() should be paired with an *exit_ro*() .
- *exit_ro*(X): Marks the end of read-only access to X.
- *fence*() : Issues a fence operation. This should prevent the compiler from reordering code and issuing proper fence instructions for an out-of-order processor.
- *flush*(X): Because an *exit_x*(X) is lazy, a flush of X forces modifications to X to become globally visible. Concurrent read-only accesses then can receive the update. This is a best-effort operation, so there are no guarantees that all processes actually observe the modifications within a specific amount of time. It is only allowed to flush an object inside a *entry_x*()/*exit_x*() pair.

When these annotations are properly applied to the example of Fig. 5, the resulting source code is shown in Fig. 6. The *flush*(f) is added to make sure that process 2 will read the value 1 eventually. A flush of X is not needed, because the acquire of X will always get the latest modifications.

The annotations are applied to shared objects of any size, which conflicts with the memory model. Recall, the memory model of Section IV assumes operations on variables of

Table II
IMPLEMENTATION ON DIFFERENT ARCHITECTURES

annotation	Software cache coherency	DSM over write-only interconnect	SPM and SDRAM
read/write	By design, the MicroBlaze implements (at least) Slow Consistency. It exhibits in-order execution and no interconnect reorders operations of one processor. So $\prec_\ell$ and $\prec_P$ between reads and writes are satisfied by the hardware.		
fence	Because the MicroBlaze is in-order, the fence only controls reordering by the compiler and does not emit any instructions. So $\prec_\ell$ and $\prec_F$ between fences and other operations is satisfied by the hardware.		
entry_x	Exclusive access is enforced by acquiring a lock on a mutex that is related to the object that is protected. $\prec_S$ is implemented using the distributed lock [15]. To ensure $\prec_P$ between the acquire and successive operations, when the lock is transferred to another processor...		
	...the object is flushed from the cache. So, the object does not reside in the cache outside of any entry/exit pair.	...the local version of the object is written to the local memory of the acquiring processor.	...the acquiring processor makes a local copy of the objects version in the SDRAM.
exit_x	Releases the lock on the object. Because the MicroBlaze is in-order, $\prec_P$ between the release and preceding operations is automatically guaranteed by the hardware.		The data is copied back to SDRAM.
entry_ro	When the size of the object is one byte, it does nothing. Otherwise, it acquires the same lock on the object as <i>entry_x</i> .		Makes a local copy of the object. If the object is larger than one byte, the object is locked before copying and unlocked afterwards.
exit_ro	Flushes the corresponding cache lines and releases the lock if <i>entry_ro</i> locked it.	Releases the lock if <i>entry_ro</i> locked it, otherwise does nothing.	Discards the local copy.
flush	Flushes the corresponding cache lines.	Makes a copy of the object in the local memory to all other local memories.	Copies the object back to SDRAM.

atomic locations, which must be just one byte. Most real-life data structures are larger than that, like a **struct** or a **double** on a 32-bit machine. In general, when such a multi-byte object is read, it is required that one protects the object with a mutex to prevent reading the new first half of the double and the old second half, for example. Hence, the compiler that processes the annotations must decide whether locking is required for read-only access. Although this decision is easy, it influences efficiency of the program.

With annotations in place (either by the programmer or a compiler), all information about the essential ordering of the application is available. Using this information, it is possible to map the application to the platform at hand.

B. Back-end Example: 32-core MicroBlaze SoC

Given the annotations of above, we claim that it is possible to map the application to any common multi-processor hardware architecture, regardless of its supported memory model. For a sequential consistent system, the implementation of the annotations is trivial; mutual exclusion is still required for the entry/exit pairs, but all other annotations can be ignored safely, because the hardware takes already care of it.

We study the implementation of the annotations for hardware that implements a weaker memory model. For this, we use a 32-core MicroBlaze system [15, 16], realized on FPGA using the Xilinx ML605 development board. It contains support to measure micro-architectural events, like

Figure 7. Distributed memory architecture, with write-only access to other's local memory

counting instructions and cache misses. Fig. 7 shows a simplified overview of the architecture. The system consists of tiles. Every tile contains one MicroBlaze and a local memory. All MicroBlazes can access an SDRAM memory via non-coherent cache. Moreover, they can also write into each others local memory via a network-on-chip (NoC).

This architecture is used demonstrate three memory models c.q. architectures: 1) a software cache coherent multi-processor system (and the local memories are not used); 2) a distributed shared memory (DSM) architecture, where all local memories are kept coherent, such that they form a shared memory (and the SDRAM is not used); and 3) a setup where the local memory is used as scratch-pad memory (SPM). At first glance, it seems not trivial to use these three completely different architectures as back-end of the same memory model. However, the implementation of the

annotations for these architectures is listed in Table II and will be discussed below. For the experiments, we designed a single C++ interface that defines the annotations, where the implementation/back-end can be changed transparently to the application.

The first setup relies on properly flushing the caches. The cache of the MicroBlaze is only capable of either invalidating dirty data in the cache or flushing dirty data and invalidating it afterwards. So it is not possible to reconcile a dirty cache line, without also removing it from the cache. All shared objects are aligned to a cache line by compiler directives and cannot overlap with other objects. The second column of Table II describes how the annotations are implemented for software cache coherency. This protocol resembles the BACKER cache coherency protocol [17].

In the DSM setup, the software must write local updates of the data to another’s local memory via a write-only interconnect. When this is done properly, all local memories hold the same data and the MicroBlazes see the local memory as one single shared memory. The third column of Table II shows the implementation to achieve this. So, although reading each other’s local memory is impossible, write-only access is sufficient to make memories coherent.

Finally, the SPM setup makes a local copy of the SDRAM for local processing. When the application is finished using the data, it is either copied back to main memory or discarded, depending on whether the data has changed. Although SPMs often require compiler support for higher efficiency, we chose to manage it at run-time, because of simplicity of the implementation.

In retrospect, the PMC memory model allows *abstraction* of the memory model of the hardware. The different implementations as discussed above show how software complements the memory model of the hardware. The next section discusses the implementation of applications on the PMC memory model and are easily portable to any of the aforementioned three architectures.

VI. CASE STUDY: PORTING TO DISTRIBUTED MEMORY

As a case study, we implement applications for PMC for the three architectures of the previous section to show the feasibility of the approach.

A. Software Cache Coherency: SPLASH-2 Benchmark

The first case study maps applications to the 32-core MicroBlaze system and focuses on adding software cache coherency transparently. Hardware cache coherency is one of the important issues that limit scalability to many cores, because of the complexity of hardware cache coherency protocols [18]. On the other hand, software cache coherency is often discarded as a viable alternative, as it requires a strongly disciplined programming approach. As a consequence, shared data is predestined to be uncached in such system. In this experiment, the annotations of Section V-A are applied to investigate the feasibility of software cache coherency.

Figure 8. Measured execution time and processor utilization of non-cached and software cache coherency

We picked three applications from the SPLASH-2 benchmark set [19]: RADIOSITY, RAYTRACE, and VOLREND. For these applications, we ran two experiments: 1) a setup where all private data (the stack, heap and data structures of the OS) is cached, but all application data that is shared between processors, resides in uncached memory; so no cache coherency protocol is required and all cache flushes are nullified; and 2) a setup where all memory is cached, so the protocol discussed above is applied.

Fig. 8 shows the performance results of both experiments, labeled ‘no CC’ for the first experiment with uncached shared data, and ‘SWCC’ for the second. For all applications, it is indicated which percentage of the total execution time is used for the actual calculations, or the processor stalls. The stalls are categorized as: a stall because of a data cache miss when reading private data, a stall on reading shared data (after a data cache miss or just an uncached read, depending on experiment), a stall on writing (hardly visible in the figure), and a stall on instruction cache miss. For example, RADIOSITY without cache coherency has an effective utilization of 38%. Applying software cache coherency improves the total execution time by 26% and the core utilization increased to 70%. So, the execution time improved by 22% on average for these applications when using software cache coherency. The time spent on executing flush instructions for software cache coherency is for the three applications respectively 0.66%, 0.00%, and 0.01% of the total run time—the overhead is negligible.

The implemented cache protocol forces shared data out of the cache during the exit call. So, executing two consecutive non-exclusive sections will read data from background memory twice. Worst case, data is flushed from the cache after every read. In Fig. 8, the stall time on reading data is separated in reading private and shared data, of which


```

1 template <typename T,int N,int R> class MFifo {
2     T buf[N];
3     int write_ptr, read_ptr[R];
4 public:
5     void push(T data){
6         int wp,rp;
7         entry_x(write_ptr);
8         wp = write_ptr%N;
9         // Wait until all readers got buf[wp]
10        for(int i=0;i<R;i++)
11            do{
12                entry_ro(read_ptr[i]);
13                rp = read_ptr[i];
14                exit_ro( read_ptr[i]);
15            }while(rp<wp-N);
16        fence();
17        entry_x(buf[wp]);
18        buf[wp] = data;
19        exit_x( buf[wp]);
20        fence();
21        write_ptr++;
22        flush( write_ptr);
23        exit_x( write_ptr);
24    }
25    const T pop(){
26        int wp,rp,me=get_reader_id();
27        entry_ro(read_ptr[me]);
28        rp = read_ptr[me]%N;
29        exit_ro( read_ptr[me]);
30        do{ // Wait until data is written
31            entry_ro(write_ptr);
32            wp = write_ptr;
33            exit_ro( write_ptr);
34        }while(wp<=rp);
35        fence();
36        entry_x( buf[rp]);
37        T data = buf[rp];
38        exit_x( buf[rp]);
39        fence();
40        entry_x( read_ptr[me]);
41        read_ptr[me]++;
42        flush( read_ptr[me]);
43        exit_x( read_ptr[me]);
44        return data;
45    }
46 };

```

Figure 9. Outline of a multiple-reader, multiple-writer FIFO in C++, with element type T, a buffer depth of N, and R readers. The essential orderings are indicated.

the latter is conservatively (e.g. over-estimated) measured. The figure shows that for RAYTRACE and VOLREND, there are hardly any stalls on reading shared data when applying software cache coherency. For RADIOSITY, the stall time is reduced, although not as much as for the other applications. This is due to the design of the application, which addresses and updates the memory in a chaotic way.

This experiment shows that it is feasible and beneficial to annotate the application and transparently apply software cache coherency.

B. Distributed Shared Memory: Multi-Reader/-Writer FIFO

The second case study uses the architecture where all local memories are used as a single software-managed distributed

```

1 // implementation of annotations (see Table II)
2 template <typename T> class ScopeRO {
3     const T& obj;
4     T* spm;
5 public:
6     ScopeRO(const T& o) : obj(o) { // entry_ro
7         spm = (T*)alloc_spm(sizeof(T));
8         if(sizeof(T)>1) lock(obj);
9         memcpy(spm,&obj,sizeof(T));
10        if(sizeof(T)>1) unlock(obj);
11    }
12    ~ScopeRO { free_spm(spm); } // exit_ro
13    operator const T&() { return *spm; }
14 };
15
16 // application code
17 typedef struct {
18     const Window* window;
19     const MBlock* mblock;
20     Vector* vector; } work_t;
21
22 Vector motion_est(const Window&,const MBlock&);
23
24 void worker(){
25     work_t work;
26     while((work=queue.pop())){
27         ScopeRO<Window> window_s(*work.window);
28         ScopeRO<MBlock> mblock_s(*work.mblock);
29         ScopeX<Vector> vector_s(*work.vector);
30         vector_s = motion_est(window_s,mblock_s);
31         // all scope objects destructed
32     }
33 }

```

Figure 10. More complex scoping support in C++, with an alternative approach to handle entry/exit pairs

shared memory system, which are all connected via a write-only interconnect. Although the SPLASH-2 applications above in theory can be mapped onto this architecture, the local memories in our system are too small to put all data in them. Therefore, we discuss another application: a multiple-reader, multiple-writer FIFO. Such FIFO in combination with distributed memory is useful in streaming applications [20, 21].

Fig. 9 shows an outline of the implementation of such FIFO. For simplicity, only push() and pop() are given and checks for an int overflow of the pointers have been left out. The figure indicates which ordering rules apply to the source code. A nice property of this implementation is that the read and write pointers are only polled from local memory, which is fast and does not influence the execution of other processors. The DSM back-end (see Table II, third column) makes sure that updates will arrive properly.

Although this example is given in the context of distributed memory, the FIFO behaves also correctly on all of the other architectures.

C. Scratch-pad Memory: Motion Estimation

The last case study shows how the PMC approach can be used for a typical SPM application: motion estimation. In video encoding, the motion of an object is used for

compression. For this, a video frame is separated in a matrix of blocks. Then, every block of the next frame is matched within a search window of a reference frame. A naive algorithm to find the motion vector is to do a full search. In such approach, it is efficient to store both the block and the search window locally, because they are read many times. In that context, an SPM can be beneficial.

There is a practical issue when dealing with an SPM when the processor does have an MMU: an object has two addresses, one of the main memory and one of the SPM. It is more convenient when the annotations hide this. We implemented several C++ classes, as an example of how such complexities can be hidden and how dealing with the memory model is better integrated in the language.

Fig. 10 gives a partial C++ implementation of a motion estimation application and the annotations for SPMs. Assume that the `worker()` function is executed by one thread, which gets work packets via a queue. Then, it accesses the search window and block, and executes the matching function to determine the motion vector. The entry/exit calls are handled by the `ScopeRO` class, where the entry call is implemented by the constructor and exit by the destructor. The implementation corresponds to the fourth column of Table II. When the `ScopeRO` object is cast on line 30 to access the actual data, a reference is returned to the SPM and the original data is left untouched. Although the concept of the annotations stays the same, this shows that it depends on the language how they can be used effectively.

Like the previous examples, the application is now independent of the underlying memory model. Although it depends on many architectural parameters, experiments show a significant performance increase when this application is using SPMs, compared to the software cache coherency setup.

VII. CONCLUSION

One of the issues of porting an application to different hardware, is a change in the memory model. This paper proposes an approach that makes applications independent of the memory model of the hardware, in order to allow transparent mapping to different platforms. It consists of a weak, synchronized memory model that defines the fundamental orderings an application can assume, and annotations that allow defining additional ordering constraints.

The memory model 1) is an intersection of all orderings of all common memory models to allow maximum ordering flexibility; but 2) is still strong enough to behave like Processor Consistency, and can therefore simulate SC for data-race free applications; 3) is weaker than Entry Consistency, because of relaxed constraints on the ordering of synchronization operations; and 4) clearly distinguished the four different types of orderings, which allows straight-forward usage. Next, annotations in the application give the tooling all information about the additional ordering requirements, such that it can automatically insert logic to complement the hardware orderings when necessary.

The case study shows that software cache coherency can be applied transparently to several SPLASH-2 applications, which benefit 22% in execution time over uncached shared data. Moreover, a mapping is discussed to two other non-trivial memory architectures, namely distributed shared memory and scratch-pad memories. Examples demonstrate how applications can be written, such that they can be easily mapped on all of these architectures.

REFERENCES

- [1] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.
- [2] B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway distributed shared memory system," in *Compcon Spring '93, Digest of Papers.*, Feb. 1993, pp. 528–537.
- [3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 18, pp. 15–26, May 1990.
- [4] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," in *Proc. of SPAA*. ACM, 1993, pp. 251–260.
- [5] D. Mosberger, "Memory consistency models," *SIGOPS Oper. Syst. Rev.*, vol. 27, pp. 18–26, Jan. 1993.
- [6] P. Hutto and M. Ahamad, "Slow memory: weakening consistency to enhance concurrency in distributed shared memories," in *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, May 1990, pp. 302–309.
- [7] G. Gao and V. Sarkar, "Location consistency—a new memory model and cache consistency protocol," *Computers, IEEE Transactions on*, vol. 49, no. 8, pp. 798–813, Aug. 2000.
- [8] G. Long, N. Yuan, and D. Fan, "Location Consistency model revisited: Problem, solution and prospects," in *PDCAT*, Dec. 2008, pp. 91–98.
- [9] M. Frigo, "The weakest reasonable memory model," Master's thesis, MIT Department of EE and CS, Jan. 1998.
- [10] R. Lipton and J. Sandberg, "PRAM: A scalable shared memory," Princeton University, Tech. Rep. CS-TR-180-88, Sep. 1988.
- [11] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *J. ACM*, vol. 51, pp. 800–849, Sep. 2004.
- [12] C++11, Std. ISO/IEC 14882:2011.
- [13] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *POPL*. ACM, 2011, pp. 55–66.
- [14] C. Wallace, G. Tremblay, and J. N. Amaral, "On the tamability of the Location Consistency memory model," in *Proc. of the Int. Conf. on PDPTA*. CSREA Press, 2002, pp. 1542–1550.
- [15] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit, "An efficient asymmetric distributed lock for embedded multiprocessor systems," in *IC-SAMOS*, 2012, pp. 176–182.
- [16] —, "Evaluation of a connectionless NoC for a real-time distributed shared memory many-core system," in *DSD*, 2012, pp. 727–730.
- [17] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "DAG-consistent distributed shared memory," in *Proc. 10th Int. Parallel Processing Symp., IPPS '96*, 1996, pp. 132–141.
- [18] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *PACT*, 2011, pp. 155–166.
- [19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *SIGARCH Comput. Archit. News*, vol. 23, no. 2. ACM, May 1995, pp. 24–36.
- [20] K. Denolf, M. J. G. Bekooij, J. Cockx, D. Verkest, and H. Corporaal, "Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations," *EURASIP J. Adv. Sig. Proc.*, 2007.
- [21] T. Bijlsma, M. J. Bekooij, and G. J. Smit, "Circular buffers with multiple overlapping windows for cyclic task graphs," in *Transactions on High-Performance Embedded Architectures and Compilers III*, ser. LNCS, P. Stenström, Ed. Berlin: Springer Verlag, Mar. 2011, vol. 6590.