A Unified Resource Model for Engineering Multi-Core Software Systems

András Vajda
Ericsson Software Research
Hirsalantie 11
FI-02420 Jorvas
+358 9 299 3046
andras.vajda@ericsson.com

ABSTRACT

Multi-core architectures and chip multi-processor systems (CMPs) have become the mainstream approach for building processors delivering improved theoretical performance according to Moore's law. From a software engineering perspective, however, this represented a major shift from HW-driven performance improvement to software-based methods. In this context, the efficient usage (in terms of development and run-time resources) of processing cores and memory is of paramount importance, as these two types of resources are at the root of most challenges related to developing parallel software for CMPs.

In this paper we present a unified resource model (URM) environment and a method for efficiently and transparently managing computing resources, building on a number of related technologies, such as model-based design and automatic code generation. This model enables the programmer to focus on the problem domain without the need to deal with shared memory access, scheduling or processor core management issues.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.1.7 [**Visual Programming**]; D.2.2 [**Design Tools and Techniques**]: Software libraries; D.3.4 [**Processors]:** Code generation, Compilers; D.4.1 [**Process Management]:** Concurrency, Multiprocessing/multiprogramming/multitasking, Threads

General Terms

Algorithms, Management, Design, Languages.

Keywords

Multi-core, Resource Management, Software Engineering, Shared Memory, Scheduling, Modeling, Code Generation

1. INTRODUCTION

Multi-core architectures and chip multi-processor systems (CMPs) have become the mainstream approach for building processors delivering improved theoretical performance according to Moore's law. From software engineering perspective however, multi-core HW technology triggered a major shift from HW-driven performance improvement to software-based methods. This paradigm shift – and its consequences for the industry – led to a substantial amount of research effort dedicated to appropriate computer architectures, operating systems and programming models.

One of the fundamental issues that need to be tackled in CMPs is the management of shared resources, such as processing cores, memory and I/O capabilities. From a multi-core software engineering perspective, the efficient usage (in terms of development and run-time resources) of the first two of these – processing cores and memory – is of paramount importance, as these two types of resources are at the root of most challenges related to developing parallel software for CMPs. Consequently, one of the key challenges is to provide a framework that hides the details of managing cores and memory and enables programmers to focus on the actual problem domain.

We will present a method for efficiently and transparently managing these resources, building on a number of related technologies, such as model-based design and automatic code generation, in the context of a unified framework for describing resources and resource application requirements.

2. RESOURCE CHARACTERIZATION

2.1 Programming Models

Fundamentally, there are four main groups of parallel programming models proposed so far, defined along two axes: parallelism model (thread or task based) and communication model (shared memory or message passing). Essentially, all models proposed so far deploy a variant of these four basic models, combined with various scheduling approaches. Table 1 shows a summary of these models.

Table 1 *Programming model paradigms*

	Parallelism:	
Communication	Thread based, shared memory	Task based, shared memory
	Thread based, message passing	Task based, message passing

We will consider all four of these models in the present paper.

2.2 Management of Processing Cores

There are two fundamental types of multi-core processors: homogeneous ones, characterized by having processing cores of equal capabilities (in terms of ISA, speed, pipeline architecture and cache structure) and heterogeneous processors, comprising processor cores with different capabilities (in terms of ISA, speed, pipeline architecture or cache architecture).

From a resource management point of view, homogeneous systems are usually managed through symmetric multi-processing (SMP) scheduling. The more challenging class of processors is of those with heterogeneous capabilities. Heterogeneity can manifest itself in terms of varying, perhaps partly overlapping ISAs (e.g. RISC architecture core combined with a VLIW DSP core), varying execution speed (frequency), varying cache sizes, varying capabilities (such as pipe-line depth). A significant body of research deals with scheduling and programming issues related to these chips, both on OS and programming model level. Due to this complexity, in this paper we will primarily focus on heterogeneous systems.

2.3 Shared Memory

One of the easiest to grasp approaches to program data is that of shared memory. Unfortunately, usage of shared memory in a CMP poses a number of daunting challenges in terms of synchronization, resource contention and memory bandwidth management that increases dramatically the cost of efficiency for using shared memory. All of the traditional approaches to shared memory – locks, transactional memory, and lock-free data structures – have a number of drawbacks:

- Locks are *non-composable*, i.e., two pieces of correct program code, when combined, may not perform correctly, leading to hard-to-detect deadlock or live-lock situations
- Transactional memory solutions, while composable, have a *significant processing overhead*, usually require HW support and software realizations *do not scale well*: the system will perform increasingly inefficiently in case the number of processing elements trying to access the same data is increased; it has been shown previously that Haskell's STM solution does not scale beyond 8 cores in a CMP ([1])
- Neither locks nor STM solutions are *predictable and deterministic*, i.e., it's very difficult – and in some cases impossible – to calculate a reliable upper-bound for execution time; this behavior is not suitable for real-time applications
- Lock-free data structures and algorithms require case-by-case development and there as no universally applicable solution available.

Recently, a new approach called LTF-SHM (Lock and Transaction Free Shared Memory) has been proposed ([3], [2], [8]), relying on the simple principle of locking shared data structures to specific cores and moving computations – whenever needed – to the core that 'owns' the data structure that needs to be addressed. As shown in [3], this approach offers a dead-lock free, deterministic, self-adaptive and simple approach to shared memory in CMPs, one that does not require any elaborate design effort by the programmer.

3. PROPOSED MODEL

3.1 Unified Resource Model

We propose a unified resource model (URM) that captures the potential heterogeneity of the hardware as well as the shared data structures into one unified framework that can be used as a basis for automatic generation of parallel software for various targets. It offers a framework for reasoning about shared data structures, or need for specialized computing architecture features (instruction sets, speed or memory size). It does not require the programmer to know in detail how the hardware looks like, rather it offers the setting and tools to express what the program will require.

We start from the observation that applications will expose their computing requirements as *requests for certain type of resources.* For example, a piece of code (function or critical section) that needs access to a shared memory area exposes a property indicating that it requires a certain type of resource, in this case, *access to a certain location in the memory*. Another such example is the need for a certain type of processor core – e.g. DSP or SPE in a Cell processor – for executing a certain part of the code, expressed, again, as a property indicating that *access to a specific core type is needed.*

Based on this observation, we propose a computing resource model where all resources are a subtype of a basic type we'll call *NeedToAccess.* In this context, a shared memory location for example can be represented as a subclass of *NeedToAccess* with the properties *id* and *type.* The requirement to execute a sequential portion of an application becomes a resource request for an object of another subclass of *NeedToAccess,* characterized by e.g. *frequency* (if frequency is the measure that defines the speed of a core). Figure 1 gives an overview of the main subclasses of *NeedToAccess*, explained in more detail below.

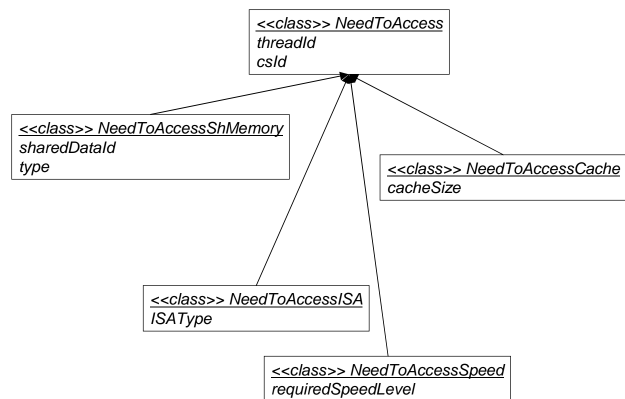

Figure 1: *Class diagram of the NeedToAccess-based sub-classes*

NeedToAccessShMemory: this type of resource encompasses the access to a certain shared memory location and it's characterized by *sharedDataId,* and *type.* It's important to stress that the focus is on *access to a specific memory area* as a resource and *not* on the amount of memory. This is a novel approach that makes shared memory areas analogous to processor cores; this can be achieved in the context of LTF-SHM, where access to a certain shared memory area means access to a specific processor core, that has the *capability of accessing* that specific shared memory area.

NeedToAcccesISA: this resource type covers cores of different types (defined by the attribute I*SAType,* which could be DSP (Digital Signal Processor), GPU (Graphics Processing Unit), GPP (General Purpose Processor) etc). Through expressing a need for this type of resource, the programmer can provide information regarding where a certain piece of code shall be executed in order

to improve the performance of the program (e.g. video transcoding is best suited for DSP type of processors).

NeedToAccessSpeed: this type of resource represents access to high performance cores, typically needed for executing sequential portions of applications. Through expressing a need for this type of resource, the programmer can provide information regarding where a certain piece of code shall be executed – but in the context of execution speed rather than specific type of functionality.

NeedToAccessCache: this type of resource represents access to cores with a defined minimal amount of local cache. This type of resource can be used in cases where a large amount of data is manipulated that is best fitted in the internal memory. The resource object may also define which memory locations will be accessed in order to facilitate pre-fetching.

These resource types are just examples we have identified as essential ones; more resource types can be defined as needed.

3.2 Programming Model

The resource model presented in the previous section provides the foundation for a high level programming framework that shields the programmer from the complexity of managing resources typical to a multi-core application.

The proposed programming framework builds on the concepts of *critical sections* and the *design by contract* paradigm. Essentially, the model requires the programmer to annotate each section of the code that have specific requirements – e.g. the need to access a shared memory location, run at higher speed or using a specific type of core – by marking it as critical section, with associated *resource contracts* in terms of objects of type subclasses of *NeedToAccess,* hence defining the type of resource the critical section requires.

For the implementation of the proposed method we will use a model-based design approach. In the context of a UML-based modeling environment, the critical section annotations can be implemented as *model markings* and the resource contracts as the values associated with the markings. To facilitate markings, each section of code defined as critical section is best delimited as a function, object method or transition (if the UML real-time profile is used). We propose the following notation:

<<CS, resourceType={objectType, attributes}, resourceType= {objectType, attributes}, … >>

Figure 2 gives a few examples of using this type of marking. Please note that there may be several resource objects associated with a critical section (e.g. for expressing the need to access several shared memory areas simultaneously).

One extension of the proposed marking model is to allow *non-binding* resource contracts, through which the programmer can express the *recommendation* to use a specific type of resource (such as ISA or speed), without making it mandatory. This approach gives the model transformation / compilation phase more flexibility in managing contracts and generating code.

```
<<method>> void CSForSharedMemory (unsigned a, unsigned b)
<<CS:
   resourcType = {NeedToAccessShMemory, DATA_1,
                  addressPointer}>>
{
  // method code
}
```

```
<<method>> void FFT (void *params)
<<CS:
   resourcType = {NeedToAccessISA, DSP} >>
{
  // method code that shall execute on DSPs
}
```

```
<<method>> void CSSequentialCode ()
<<CS:
   resourcType = {NeedToAccessSpeed, "2GHz"} >>
{
  // method code that shall execute at high speed
}
```

Figure 2: *Examples of markings*

4. IMPLEMENTATION

The fundamental reason for choosing a modeling based approach (and particularly, a UML based one) is the framework it offers for platform independent design in the form of Computation Independent Models (CIM – implementation independent representation of the required functionality) and Platform Independent Models (PIM), derived from CIMs and coupled with one or several model compilers that can automatically generate, based on the model, annotations and specific deployment models platform specific models (PSM), suitable for a specific target hardware and software systems. These concepts are illustrated in Figure 3. For more information, please see [9].

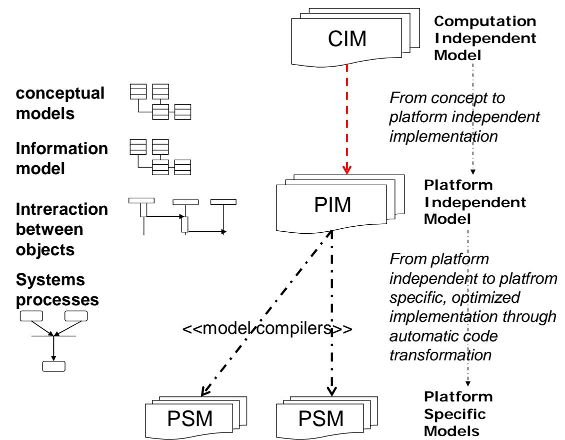

Figure 3 *The concepts of CIM, PIM and PSM*

The essence of model-based development is that software shall be developed in a platform independent fashion, in the problem domain (such as telecommunication, scientific computing etc) and then mapped through model transformations (or compilation) to specific, potentially multiple targets. The target-specific information is captured in the model compiler and the associated run-time system (RTS), with potentially different compilers and run-time systems developed for each target. In our case, the PIM is built by the programmer relying on the universal resource model, that is itself target-independent: it only offers a framework

for reasoning about shared data structures, need for specialized computing architecture features (instruction sets, speed, memory size) etc. The key to an efficient deployment is how the model compilers and run-time systems are built. We will detail our proposed model and model compiler functionality in the following sub-sections.

4.1 Run-time System

The model at the foundation of the run-time system is based on the *task-based programming paradigm*. At a very high level, critical sections are mapped to *tasks* that will either be dispatched to specific processor cores with special roles or will be used to trigger a specific action in the operating system.

4.1.1 Run-time System for Shared Memory

Our run-time system for shared memory is based on the method described in [3], built on the concept of moving computation instead of data. Essentially, it requires the programmer to mark explicitly each memory area that is shared between multiple threads, associate a unique id (called *shared data id – SDI*) to each area and include this information – as instances of the *NeedToAccessShMemory* class – in the resource contract of the critical section that will access one or several of the shared data areas. In the run-time system, the critical sections will be dispatched as *tasks* that will be executed on specific cores (called *resource guardians*) that own the shared data areas accessed by the critical section. The method is depicted in Figure 4, reproduced from [3] (UPE means User Processing Entity, the original core where the program was executing before entering the critical section).

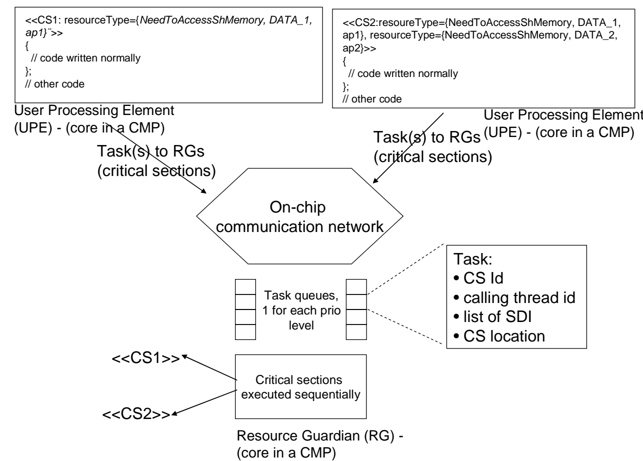

Figure 4 *Execution model for shared memory access*

4.1.2 Run-time System for ISA handling

The run-time system for this type of resource will depend to a large extent on the underlying HW and operating system structure. For example, a possible solution for the Cell BE processor has been presented in [4], relying on much the same mechanisms as our proposed critical section model.

We promote – if the underlying HW allows it – a model-based on *Remote Procedure Call (RPC)* mechanisms. In this model, the run-time system will manage a *shadow thread* for each thread that has associated critical sections that require a specific ISA (Instruction Set Architecture). The shadow thread will execute on a core having the type of ISA that the critical section requires and will get triggered through the RPC mechanism whenever the critical section needs to execute.

The model compiler shall generate separate source code files for both architectures (main and required) as well as supporting build and link files.

4.1.3 Run-time System for Executing Sequential code

The implementation of the run-time system for this kind of resource (*NeedToAccessSpeed)* depends largely on the underlying hardware.

If the hardware is homogeneous with no support for frequency or voltage scaling, the only possible implementation is to boost the priority of the thread expressing the need for this resource.

In case of asymmetric, homogeneous ISA HW (equipped with cores having the same ISA, but different performance characteristics, such as execution speed), this resource shall trigger migration of the thread to the more powerful core, where the critical section can execute in shorter time. An interesting approach based on voltage and frequency scaling can be implemented based on the methods described in [6] or [5]. In these solutions, some cores will be shut down and the frequency at which the core executing the critical section is run is temporarily boosted to the maximum supported level, thus supporting the faster execution of the critical section.

An interesting alternative for the future could be to deploy speculative, run-ahead, multi-path execution on a massive scale, based on the design by contract paradigm ([10]). At present however, no such system has yet been proposed.

4.1.4 Run-time System for Cache Sizing

The run-time system for this resource type is very similar to the *NeedToAccessSpeed* type of resource. The thread will be migrated, if possible, to a core with larger L1 or L2 cache size, but, in addition, the run-time system may perform a pre-fetching of the memory locations indicated in the resource object. Paper [8] describes a method how such a pre-fetcher could be implemented in HW, but essentially the same result can be obtained through software triggered DMA (or equivalent technology).

4.2 Model Compiler

The model compiler is the entity responsible for transforming the (annotated) UML model to a program (usually in C/C++ or Java, rarely in assembler) that fulfills two conditions:

- It is functionally equivalent to the original model
- It is specific to the HW and SW target (RTS) the model compiler is designed for

In this section we will outline the required functionality in order to successfully and efficiently map the high level URM-based annotated model to the underlying run-time system described in the previous section.

4.2.1 Shared Memory

The model compiler is responsible for defining the critical section groups (the group of critical sections that are accessing related

memory areas, see [3] for details) and generating the code for the associated resource guardian functionality, as well as mapping the resource guardian roles to available processor cores. The model compiler shall also generate the glue code for linking in the automatic dead-lock resolution and elastic resource guardian scoping functionality as well as for the dispatch of tasks on user-processing cores (the cores that execute regular, non-critical section code).

4.2.2 ISA handling

The model compiler is to a large extent dependent on the underlying HW architecture and the run-time system. In our favored model, based on RPC and shadow threads, the model compiler shall

- Identify threads (potentially based on separate programmer annotations) that have critical sections requiring different ISA resources
- Perform a static schedule analysis for required configuration; if static schedule analysis is inconclusive, code for run-time decision whether a *non-mandatory* resource ISA requirement can be fulfilled will be generated
- Generate *shadow thread* code and code for dispatching RPC tasks

We would like to emphasize that this is just our preferred approach, verified in a number of telecom applications; however, several other methods have been proposed, e.g. [4].

4.2.3 Sequential code execution

Managing the resource need for fast execution of sequential code will, again depend on the actual HW infrastructure. For asymmetric, homogeneous HW architecture, the model compiler shall generate code for migrating threads between cores (to and from the complex core); solutions for how to perform this have been proposed in e.g. [7]. In contrast to prior work, our proposal is deterministic, in the sense that it's precisely delimited (through the use of the critical section concept) the part of the code that shall execute on the higher performance core.

Our preferred solution is the one based on aggressive DVFS, as outlined in [5]. In this model, the model compiler shall generate code for

- Requiring shut-off of unused cores
- Move of the execution to a core with frequency scalability
- Requesting boost of frequency on that core
- Requesting return to normal frequency after the critical section completes

4.2.4 Cache-size resource

The model compiler shall, in this case, generate code for

- Thread migration to large sized cache cores for the period of execution of the critical section
- If the CS has a property indicating the required memory areas, code for pre-fetching the data on the target core (again, HW and OS dependent)

5. DISCUSSION

Managing resources such as memory and processing cores in the presence of massive parallelism is one of the fundamental issues of multi-core software engineering. Another aspect, critical in any industrial environment, is to increase the portability of software and re-use across a broad range of hardware platforms – this requirement is even more critical due to the cost of porting across heterogeneous architectures.

Our proposed resource management framework and programming model aim to address exactly these issues. The resource model we propose provides a simple, HW-independent, unified framework for reasoning about the major types of resources in a multi-core chip. It's important to note that indicating the requirement for a specific type of resource does not imply a need to know the underlying HW capabilities – the model compiler has the freedom to provide the semantic in a different way: shared memory using messaging in the run-time system (if there's no cache coherency support), execution of specific ISA-typed code segments on other than indicated types of HW etc. De-coupling the *reasoning on resources* and *the actual implementation* in a constrained HW/OS infrastructure is one of the strengths of our proposal.

Another benefit of the programming model that we propose is that it does not dictate neither a threading based nor a task based parallelism model, this being left to the programmer. It only provides a method for managing essential resources – the implementation of parallelism on high level is still up to the programmer. As future work, this could be brought into the same framework; however, in our experience this yielded limited benefits.

A modeling based approach also proved to be a good choice, as it allowed us to easily decouple application logic from underlying HW and OS infrastructure and from the transformation logic, while keeping all these in one unified model. By modifying small parts of the model compiler – while keeping the application model untouched – it's possible to easily experiment with various deployment strategies (auto-generated locks, transactional memory, DVFS etc).

Regarding performance, we found that the major trade-off is at finding the right run-time system (and coupled model compiler) for each HW target. The Unified Resource Model's impact on performance – versus hand-crafted code – is essentially under measurable threshold. The run-time system choices we outline in this paper are the most promising approaches, in our opinion, for main-stream massively multi-core processors, at least for telecom domain software.

6. SUMMARY AND FUTURE WORK

In this paper we presented a unified resource management framework for dealing with critical resources – such as shared memory, processor cores and cache – in a platform independent way. We couple this model with a model-based design approach to provide a simple to use, generic yet efficient method for engineering multi-core software.

As next steps, we aim to validate our model for more problem domains and various HW infrastructures. Also, extensions with more resource types will be investigated, such as parallelism model and communication resources.

7. ACKNOWLEDGMENTS

We would like to acknowledge the help of those who provided preliminary feedback on this paper.

8. REFERENCES

[1] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T. and Valero, M. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. *Proceedings of the 2008 conference on Computing frontiers* (2008)

[2] Suleman, M.A., Mutlu, O., Qureshi, M.K., Patt, Y.N. Accelerating Critical Section Execution with Assymetric Multi-Core Architectures. *In International Conference on Architectural Support for Programming Languages and Operating Systems,* (March 2009)

[3] Vajda, A. Handling of Shared Memory in Many-core systems without Locks and Transactional Memory. *3rd Workshop on Programmability Issues for Multi-core Computers (MULTIPROG),* with HiPEAC 2010

[4] Cooper P., Dolinsky U., Donaldson, A.F., Richards, A., Riley, C., Russell, G. Offload – Automatic Code Migration to Heterogeneous Multicore Systems. *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers,* (January 2010)

[5] Vajda, A. Space-shared and Frequency-scaling Based Task Scheduler for Many-core OS. *Workshop on Power Aware Computing and Systems 2009 (HotPower'09),* with SOSP 2009

[6] Greskamp, B., Karpuzcu, R.U. and Torrellas, J. LeadOut: Composing Low-Overhead Frequency-Enhancing Techniques for Single Thread Performance in Configurable Multicores. *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture,* (January 2010)

[7] Li, T., Brett, P., Knauerhause, R., Koufaty, D., Reddy, D., Hahn, S. Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture,* (January 2010)

[8] Vajda, A. The case for coherence-less distributed cache architecture. *Workshop on Chip Multi-Processor Memory Systems and Interconnect (CMP-MSI),* with HPCA 2010

[9] OMG Model Driven Architecture, http://www.omg.org/mda/

[10] Vajda, A., Stenström, P. Semantic Information Driven Speculative Execution. *Workshop on New Directions in Computer Architecture,* with MICRO 2009