Issues on Memory Managemenet for Component-based Systems

Marius Bozga, Emmanuel Sifakis

▶ **To cite this version:**

Marius Bozga, Emmanuel Sifakis. Issues on Memory Managemenet for Component-based Systems. EC2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly, Jul 2010, Edinburgh, United Kingdom. hal-00561770

HAL Id: hal-00561770
https://hal.science/hal-00561770

Submitted on 1 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Issues on Memory Management for Component-Based Models

Marius Bozga Emmanuel Sifakis
University Grenoble 1 - CNRS - VERIMAG
Centre Equation, 2 av. de Vignate, 38610 Gieres, France

1 Introduction

Memory management has become an extremely important factor for designing and implementing performant and energy efficient embedded applications. There are multiple reasons closely related to the execution platform. For instance, the memory organization and memory model enforced by the platform influence directly the execution time for any basic memory operation [1]. In addition, the amount of available memory may also directly enable/limit the concurrency allowed by particular applications e.g., pipeline or data-flow applications requiring buffering of intermediate results [2]. Finally, a considerable impact is due to the programming and memory model used at the application level. The memory model together with the programming primitives used to coordinate concurrent execution (e.g., locks, atomic sections, transactions, etc.) will heavily influence the compilation and deployment of the application onto the platform [3], [**?**], [**?**]. In particular, the placement of data into memory, the insertion of additional memory copy operations, if needed, etc are usually decided at compile time.

In this paper, we investigate these issues for the BIP – Behaviour, Interaction, Priority – component framework for modeling, analysis and implementation of heterogeneous real-time systems [4]. BIP supports a component construction methodology based on the assumption that components are obtained as the superposition of three layers: (1) behavior, expressed in terms of extended automata, (2) interactions, describing the cooperation between actions of the behaviour and (3) priorities, rules specifying scheduling policies for interactions. Layering implies a clear separation between behavior and architecture (connectors and priority rules).

At the lower level of BIP, atomic components contain behaviour characterized by a finite control described by automata and extended with arbitrary computations (expressed as C/C++ functions/methods) on arbitrary data structures (instances of C/C++ data types). Automata transitions are triggered by ports, that are, actions names used later to specify interactions. Moreover, ports may be associated with local data of atomic components. These data are available for use (i.e., reading or writing) when interactions involving that port are executed. Interactions are specified in connectors as sets of ports and have also associated an arbitrary computation involving port's data (expressed as C/C++ functions/methods). They can be executed when all atomic components involved are ready to interact i.e., every component reaches some control location enabling a transition labeled by the required port. Whenever enabled, the execution of an interaction is done in two steps: first, the interaction code is executed as an atomic step, then all involved components execute (concurrently) the local computations of the interacting transitions. When several interactions are enabled for execution, the choice is restricted according to priority rules.

Despite the use of arbitrary C/C++ for describing data types and computations, BIP tacitly assumes a private memory model for atomic components. Data are private/local to atomic components and never shared between them. This assumption confers several advantages for constructing and reasoning about systems in BIP. First of all, it achieves a separation of concerns between local computation and interaction/communication between components. This is a basic issue for reliable component-based design, as it allows to develop atomic components independently of each other, and independently from the coordination glue, once the interfaces (ports and associated data) have been clearly identified. Second, the private memory model provides scalable system analysis using compositional methods. For example, global deadlock-detection can be breakdown into local analysis of atomic components plus analysis of the interaction glue, as explained in [5]. Third, a private memory model ensures maximal parallelism between atomic components at execution. Once an interaction is completed, local computations can be carried out independently on atomic components with no interference. In particular, this model may lead to efficient distributed implementations [6].

Nevertheless, the advantages obtained from the private memory model come with a potential overhead for communication of data between components. In the absence of data sharing, successive operations on the same data require an explicit deep copy/transfer of data, if the operations are implemented on different atomic components. Such data transfers must be actually realized on BIP interactions, however, they can be very time consuming, in particular for big/complex data structures. In addition, having long computations taking place on interactions may drastically decrease the parallelism within the system. This is because all the components involved in an interaction must wait for it to complete, before resuming their internal computations. Finally, another inconvenience of the private memory model is the overall amount of memory needed for implementation. In fact, providing all the memory to ensure maximal parallelism for atomic components is not justified/needed if the execution platform underneath does not allow, or has very limited support, for parallel execution. Intuitively, a sequential execution platform (e.g., one core) implicitly reduces the parallelism within the BIP system to one executing component at a time. Therefore, the implementation can be realized in principle with less memory and still deliver the same time performances.

We have investigated a relaxed memory model for BIP allowing to address the runtime issues above, while partially preserving the advantages of the private memory model. In the new model, called SM-BIP (for Shared-Memory BIP) atomic components are allowed to own and use, temporarily, data placed in a designated/predefined Shared-Memory component. In addition, atomic components are allowed to exchange ownership/references of shared memory data on interactions. Consequently, the overhead for communicating data placed in shared memory is drastically reduced. Moreover, only one atomic component may own (i.e., points-to, reference) some shared data at a time. Hence, there are no access conflicts on the shared data, and all components can

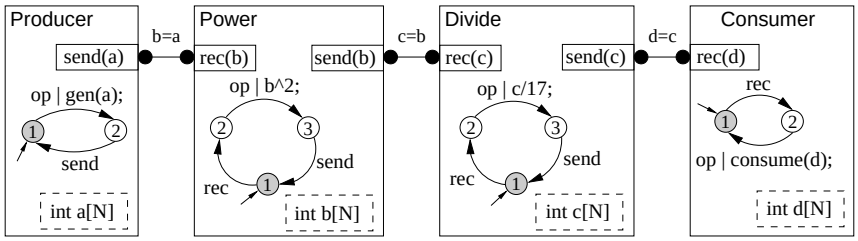

Figure 1: Pipeline example in BIP

operate concurrently. Finally, the size (e.g., number of data items) of the Shared-Memory component becomes a parameter of the system and can be chosen accordingly to extra constraints.

We present hereafter a generic transformation of a significant subset of BIP into SM-BIP. This transformation preserves the functionality of the systems, while systematically increasing the time performance. Moreover, the resulting SM-BIP model obtained can be further tuned depending on the target execution platform. The size of the shared memory is now a model parameter that can be adapted to match the degree of parallelism exhibited by the system on the execution platform. That is, one can easily find the minimal/optimal memory size allowing the system to deliver its functionality with increased time performance on the chosen execution platform.

The paper is structured as follows. In section 2, we present the rules of the transformation, from BIP to SM-BIP systems, and we illustrate them on a simple example. In section 3 we provide some initial experimental results obtained for the example. Finally, we conclude with a broad discussion on the tradeoffs of the transformation in section 4.

2 From BIP to SM-BIP

We briefly introduce the transformation of BIP systems into equivalent SM-BIP systems. For the sake of clarity, we will illustrate the transformation on a simple example, shown in Figure 1.

Example 1 *The example describes a pipelined operation on data realized by four concurrent components. Each component has a private array of size* N*. The left-most component is the* Producer*. It generates fresh data (i.e., the array content) which is then sent and processed successively by the mid components in the chain, and finally discarded by the right-most component, the* Consumer*. Data are entirely copied from the left to the right component through connectors.*

The starting point of the transformation are BIP systems annotated with information about which local data is *share-able* or not. The annotations are currently provided by the user, however, they can also be defined automatically depending on higher-level information e.g., the type or the concrete access to data in the system. The transformation is applicable for annotated BIP systems that satisfy a number of simple (syntactic) feasibility conditions. These conditions ensure that local data meant to be shared are accessed and communicated in some restricted way in the original BIP system. More precisely, the BIP system must behave like an I/O system with respect to share-able data. In I/O systems, fresh data are received on inputs ports, processed locally by the component, and finally delivered through output ports. Once a component delivers the modified data, it no longer accesses it. Here are the syntactic checks in more details:

1. All connectors connected to an input port, copy data to the component (i.e., the component receives). Dually, all connectors connected to the output port, copy data from the component (i.e., the component sends). This condition is a prerequisite to perform an accurate live data analysis in atomic components, as required by the second condition below.
2. To validate every component behaves as an I/O system on its shared data, we check that every share-able data is locally dead, both after being sent and before being received on any port. Live and dead data are computed locally, on each component, according to the classical definition used in compiler optimization see e.g., [7]. If the condition holds, it will further allow to replace the deep copy/transfer of data on connectors with a reference copy and a transfer of the ownership.
3. On every connector, share-able data is only copied (i.e., neither tested nor modified), moreover, it is copied from one source to one destination component. In combination with the previous conditions, this condition allows to safely replace the deep copy of share-able data in connectors with a simple reference copy and implicitly with the transfer of ownership.

Example 2 *The pipeline example is an I/O system. Let us assume the user annotated all the local arrays to be share-able. The conditions we just stated hold, and they can be easily checked. In Figure 1, we have grayed the states where arrays are locally dead.*

If the BIP system satisfies the condition above, its transformation to SM-BIP is structural. The overall architecture of the BIP system is preserved, however, atomic components and connectors are locally modified.

First of all, the SM-BIP system contains a Shared-Memory component. This component is a simplified memory manager which manages several memory slots e.g., one for each share-able data of the initial BIP system. For each slot, it maintains a boolean value indicating if the slot is currently in use by a component or not. An abstract view of the Shared-Memory is given on Figure 2. It can interact on two ports, alloc and free through which regular components can acquire and release memory slots. For allocation, if there are slots available, the Shared-Memory marks one of them as used and provides a reference of it to the component. Otherwise, the interaction is postponed until memory slots become available. For release, atomic components simply provide the reference of the slot and the Shared-Memory marks them as unused.

Second, atomic components and connectors are locally transformed, according to the following local rules:

- in every atomic component, every share-able data is replaced by a reference (i.e., pointer), while the data themselves are migrated as memory slots into the new Shared-Memory component.

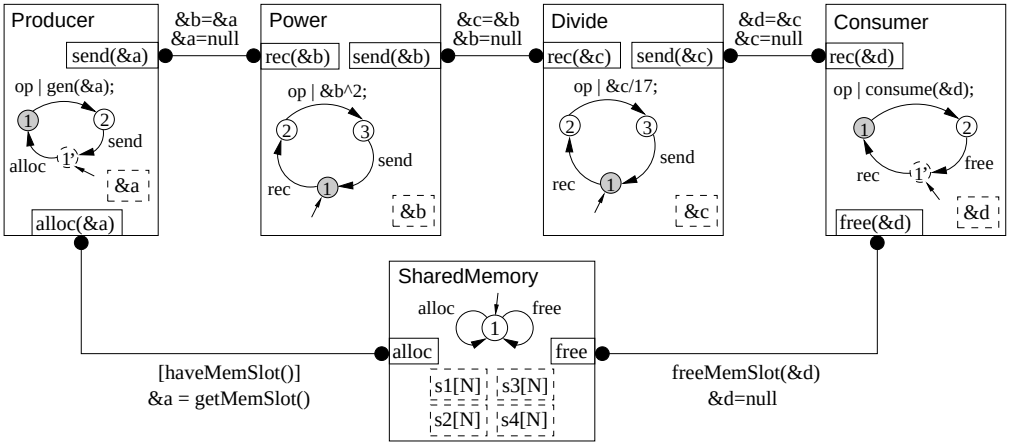

Figure 2: Pipeline example transformed in SM-BIP

- in every connector, any deep copy/transfer of share-able data is replaced by the copy of the reference. Moreover, the sender component erases its reference, that means, the ownership on the data is uniquely maintained by the receiver.
- immediately before a share-able data becomes locally live by executing any but a receive transition (e.g., the case of the Producer component), an extra state and an extra alloc transition is added to request a corresponding memory slot from the Shared-Memory.
- dually, immediately after a share-able data becomes locally dead by executing any but a send transition (e.g., the case of the Consumer component), an extra state and an extra free transition is added to release the memory slot to Shared-Memory.

These transformation rules guarantee that at any moment, every component holds unique valid references to memory slots into the Shared-Memory for all its share-able data which are actually live, and respectively null references for the share-able which are dead. Consequently, if all share-able data are entirely migrated to the Shared-Memory component, the functionality of the original BIP system is completely preserved in the transformed SM-BIP system.

Example 3 *Applying the transformation rules above to the pipeline example we obtain the equivalent SM-BIP system, depicted in Figure 2. We can notice the addition of the* Shared Memory *component, as well as the extra states of the* Producer *and* Consumer*, and the modified connectors.*

The centralized control of shared memory enhances the configurability of the system in terms of memory usage. In the SM-BIP implementation we can easily add or subtract memory slots. Altering the available memory comes with tradeoffs on parallelism and system functionality. These issues will be discussed in more details later.

3 Experimental Results

The BIP and SM-BIP implementation of the pipeline example presented earlier have been executed on an Intel Core2 Duo CPU at 2.4 GHz . The size of the arrays has been fixed at 10^6 integers. The number of iterations of the pipe has been also fixed at 100. We consider two experimental situations, respectively *light* and *heavy* computation. In the light case, each component in the pipeline makes one pass over the arrays elements, while in the heavy case each component makes two passes.

Figure 3 depicts a plot with the execution time of BIP and SM-BIP with all the possible numbers of memory slots available. The plotted data are presented on Table 1. Table 1 contains two sub-tables, one for each situation, light and heavy. The data presented on each table are, the execution time in seconds, virtual memory needed in bytes, and the percentage of the two cores utilization (the max being 200%).

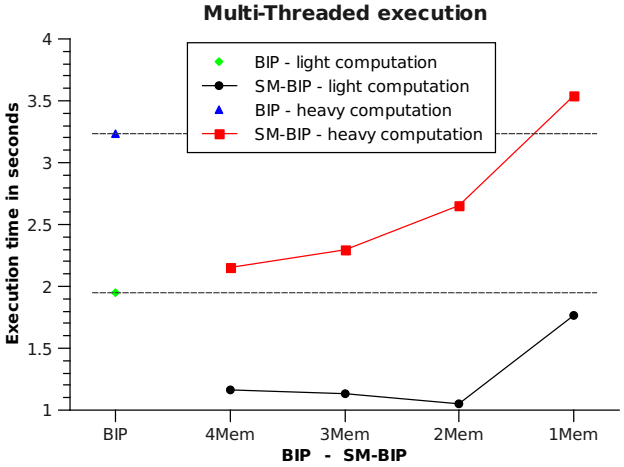

Figure 3: BIP *vs* SM-BIP, light and heavy computation.

The performance gain of SM-BIP *vs* BIP is due to replacing deep copies of the data by copy of references. For this example, the overhead of copying data was approximately 1 second. From the plot of both light and heavy computation we can observe that the SM-BIP system with the same number of memory slots (i.e., 4) as the initial BIP system, performs better. In this case, no parallelism is lost.

For this particular example, if we reduce the number of memory slots available, we simply restrict the number of parallel processing in the pipeline to the number of available memory slots. Hence, the overall functionality of the pipeline is 'preserved' to some extend. In particular, the system preserves its maximal degree of parallelism on two cores using 2 memory slots.

Nevertheless, although we systematically gain 1 second from memory copies, the lack of parallelism starts impacting the overall performance. As we can see in the case of heavy computation with one memory slot, i.e. sequential behavior, the gain due to memory copies cannot compensate the loss

(a) Light computation

		Time	Mem usage	Core utilization%
BIP	4 Mem	1.95	51672 B	143%
SM-BIP	4 Mem	1.16	59876 B	153%
	3 Mem	1.13	55968 B	157%
	2 Mem	1.05	52060 B	165%
	1 Mem	1.77	48152 B	97%

(b) Heavy computation

		Time	Mem usage	Core utilization%
BIP	4 Mem	3.23	51672 B	141%
SM-BIP	4 Mem	2.15	59876 B	174%
	3 Mem	2.30	55968 B	161%
	2 Mem	2.65	52060 B	136%
	1 Mem	3.54	48152 B	99%

Table 1: Experimental results

in parallelism, hence the execution time is worse than for the initial BIP model.

4 Discussion

We presented a transformation of BIP systems into equivalent SM-BIP systems, which always increases the time performance by replacing the costful deep copies with reference passing. Moreover, managing the shared memory slots through a component gives us the flexibility to alter the number of slots without modifying the rest of the model. In general, increasing the number of slots is safe. That is, the systems behavior remains the same. Nevertheless, decreasing the number of memory slots has drawbacks, although it might be necessary on systems with limited resources. For example, it reduces parallelism, since now components can be blocked while waiting for memory to be freed. Consequently, the systems behavior can be severely disturbed (restricted) e.g., deadlocks can occur due to unavailable memory to complete ongoing computations.

Figure 4 gives an overview of the trade-off between memory and parallelism used for implementation of SM-BIP models. On the horizontal axis we display the shared memory, in terms of slots, while on the vertical axis the number of processes that can operate concurrently on shared data. Concerning the memory, there exists actually three bounds, n_{max}, n_{real} and n_{dlk}. The upper bound n_{max} is the memory allocated by the initial BIP model, i.e., the amount of memory allowing for maximum parallelism. The second bound n_{real} is the number of memory slots that are effectively used in parallel within the initial BIP model. In fact, the BIP model itself can restrict the parallel usage of memory, due to specific synchronizations for instance. The lower bound n_{dlk}, is the minimum amount of memory needed for the system to continue to work without deadlock due to memory limitations i.e., to still deliver some (minimal) functionality.

Between n_{max} and n_{real} the SM-BIP model is equivalent to the initial BIP model. For sizes between n_{real} and n_{dlk} the SM-BIP model simulates the BIP model, in particular the degree of parallelism decreases. For each memory size n, we can actually obtain the minimal number of processes p_n that are able to work concurrently on shared data, at any time. Based on these values, we deduce a rough estimation of the ratio memory/parallelism needed for optimal execution of shared-memory BIP system. For instance, using memory size n will perform optimally on p_n cores. If the given execution platform provides more than p_n cores, the extra cores may not be used due to memory limitations. Dually, having less than p_n cores will actually restrict the parallelism, much more than the memory.

Finally, for memory sizes smaller than n_{dlk}, deadlocks can occur because of insufficient memory. The system cannot work anymore, unless an extra execution control is applied (e.g., a schedule that can coordinate memory allocations to avoid deadlock situations).

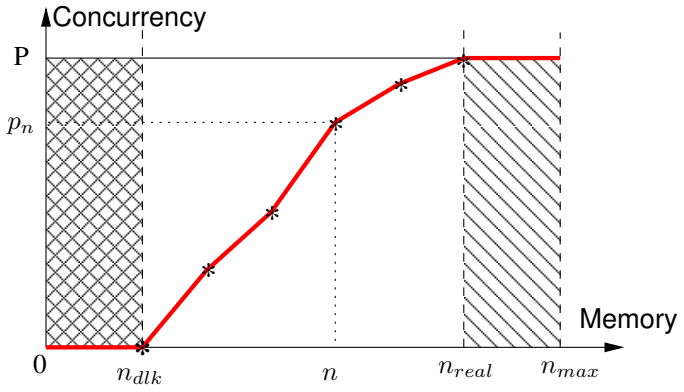

Figure 4: Reducing memory resources in SM-BIP

Interesting future work is to identify systems different than I/O to which this transformation can be applied. Furthermore, lock-based or transactional memory based solutions can be investigated for managing concurrent access to shared memory locations in component-based systems.

References

[1] S. Mador-Haim, R. Alur, and M. Martin. Plug and play components for the exploration of Memory Consistency Models. In *FMCAD09*.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv. 1995*.

[3] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. In *LCTES '09*.

[4] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream compilation for real-time embedded multicore systems. In *CGO '09*.

[5] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J.M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *PLDI '97*.

[6] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *SEFM06*.

[7] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis. Compositional Verification for Component-based Systems and Application. In *ATVA 2008*, volume 5311 of *LNCS*.

[8] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed Semantics and Implementation for Systems with Interaction and Priority. In *Proceedings of FORTE'08*.

[9] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.