Composable Virtual Memory for an Embedded SoC

Cor Meenderinck*, Anca Molnos
Delft University of Technology
Delft, the Netherlands
{a.m.molnos}@tudelft.nl

Kees Goossens
Eindhoven University of Technology
Eindhoven, the Netherlands
k.g.w.goossens@tue.nl

ABSTRACT

Systems on a Chip concurrently execute multiple applications that may start and stop at run-time, creating many use-cases. Composability reduces the verification effort, by making the functional and temporal behaviours of an application independent of other applications. Existing approaches link applications to static address ranges that cannot be reused between applications that are not simultaneously active, wasting resources. In this paper we propose a composable virtual memory scheme that enables dynamic binding and relocation of applications. Our virtual memory is also predictable, for applications with real-time constraints. We integrated the virtual memory on, CompSOC, an existing composable SoC prototyped in FPGA. The implementation indicates that virtual memory is in general expensive, because it incurs a performance loss around 39% due to address translation latency. On top of this, composability adds to virtual memory an insignificant extra performance penalty, below 1%.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*

General Terms

Design, Verification

Keywords

SoC, Composability, Predictability

1. INTRODUCTION

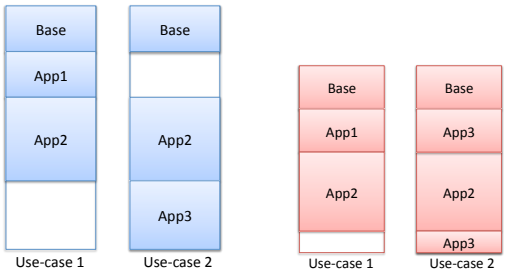
Modern multiprocessor systems on chip (SoC) concurrently execute multiple applications that can be started and stopped independently at run-time, creating many use-cases. Often, some of the applications have real-time constraints, and have hence to be *predictable*. Such SoCs typically comprise processor cores, peripheral hardware blocks, a distributed memory hierarchy, and an interconnect infrastructure. Each processor accesses a fast and relatively small local memory and a set of larger and slower memories shared among the processors. Embedded systems are cost-constrained, and resources, such as memory, must be allocated frugally, both at design time and at run-time. Typically, each use-case is statically reserved a set of (virtual) resources, and at run-time, when use-cases switch, the reserved resources are bound to the physical platform.

To reduce design costs, existing, previously implemented functional parts such as hardware blocks, application source code, application-dependent schedulers, and power management strategies are re-used. However, after integrating these parts in a larger system, their behaviours have to be re-verified [16]. This is not scalable, since a change in any system component requires re-verification of the entire system. To address this problem, *composability* [16, 23, 2] has been proposed, to allow applications to be verified independently, without requiring re-verification after integration in a larger system. A system is composable if the functional and temporal behaviours of an application are independent of the behaviour of other applications that may run concurrently.

Application development involves developing source code, optimising it, compiling it into an object file, and linking several object files into an executable that is loaded and run on the hardware. Current composable platforms allow applications to be developed independently only up to the source code level, after which they are linked in a single executable. This limits the scope of composability to loading and executing at run-time. Furthermore, statically linking all applications that can run on a processor in one executable results in position dependent code which prevents uses-cases to be loaded dynamically. Hence local memory is wasted when not all applications will ever run at the same time. As illustrated in Figure 1(a), currently a single static worst-case executable is created. This executable includes all applications that can run on the processor, leading to a waste of memory space.

Virtual Memory (VM), implemented by a Memory Management Unit (MMU), was initially conceived to allow a task to access a larger memory than the available one and to implement memory protection. It also offers position independent code, which in turn enables dynamic application loading. VM divides the memory space in pages, which reside on a large storage space, e.g., a disk. Frequently accessed pages are cached in the main memory. The page-table is typically a large data structure which stores the virtual to physical address translation, and the location of a page (memory or disk). MMUs cache the page-table in a smaller hardware structure, the Translation Look-aside Buffer (TLB), to

*Currently with Intellimagic, the Netherlands.



(a) Without VM. (b) With VM.

Figure 1: Single executable for worst-case use-case vs. executable per application when using Virtual Memory (VM).

speed-up the address translation.

Traditionally, real-time embedded systems that have to respond to events in a short, bounded period of time, do not implement VM. The VM's cache-like behaviour leads to unpredictable performance. Moreover, its overhead may be large, e.g., page faults may have a penalty of millions of cycles, which may render the system non-responsive for an unacceptably long time. However, the benefits of VM, i.e., large virtual address space and memory protection, are recognised in embedded systems that implement relatively large applications, hence predictable VM schemes are proposed. To bound the penalty of TLB misses and page faults, these schemes use special hardware page-tables [25], fixed page swapping points [12, 14], or page management and locking API [4]. However, an application may swap TLB entries or pages of other applications; applications may hence interfere with each-other. As a result, existing VM approaches are not composable.

In this paper we propose a composable virtual memory scheme that allows an application to be developed, optimised, compiled, and linked into an individual executable file. This executable file can be utilised in different use-cases, and even on different processors (with the same instruction set architecture) in the platform. We assume that the local memory suffices for each individual use-case. By using our VM the entire set of use-cases of an SoC can access a larger local memory than the one available to a processor. Virtual memory enables dynamic page allocation and virtual-to-physical address binding to reduce the memory footprint from a worst-case over all applications to a worst-case per use-case (Figure 1(b)).

Our scheme utilises conventional MMU mechanisms, such as TLB and page table, however we propose four distinguishing characteristics that makes the VM composable, predictable, and low cost, as follows. First, the VM comprises one page-table per application, as opposed to one page-table per task. In general, embedded applications have a much smaller memory footprint than desktop or server applications, hence this option is acceptable, and, even more, beneficial because it may reduce the size of the page-table. Second, the page-table of the application that runs on a processor at a given moment is entirely stored in hardware, to minimise the performance penalty of a TLB miss. Third, a light-weight Operating System updates the content of the hardware page-table and clears the TLB, at each application switch. After such switch, an application experiences a number of (cold) misses that is independent of the TLB behaviour of other applications. The penalty of these cold misses is not large, as page tables are stored in hardware,

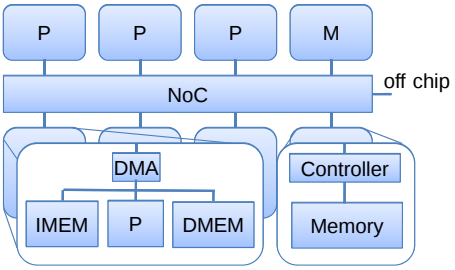


Figure 2: A typical SoC architecture consisting of processor and memory tiles.

and the access delay to these tables is hence only few cycles. Thus an application cannot swap TLB entries or pages of other applications, and cannot hence interfere with other applications. As a result, the temporal behaviour of an application is not dependent of other applications, i.e, the VM scheme is composable. Lastly, no page faults can occur, as each use-case fits in the local memory. This has two advantages, namely that the large page fault overhead is avoided, and the VM is predictable, given that the TLB behaviour of real-time applications is predictable.

The experimental platform consists of a composable SoC that is prototyped in FPGA [2] and includes MicroBlaze cores. On this platform we configure the existing MMU of a MicroBlaze core according to the scheme described above, such that it implements a composable VM. Experiments on a platform instance comprising multiple MicroBlaze cores that execute a JPEG and a synthetic application indicate composability. The MicroBlaze VM increases the instruction fetch and load/store instructions delay from 1 to 3 cycles, leading to a performance penalty of 39% for the JPEG application. Furthermore, the extra performance penalty due to the composability implementation in the VM, i.e., the TLB flush at application switch, is less than 1%, thus negligible.

The outline of this paper is as follows. Section 2 presents the target hardware and software platform. Section 3 details the composable VM, and Section 4 presents the implementation of this VM on a MicroBlaze core. Further, Section 5 presents the experimental results, Section 6 discusses the related work, and Section 5 concludes the paper.

2. BACKGROUND

In this section we present the template of the targeted multi-core platform, starting with the general hardware architecture and followed by the software infrastructure, and finally we discuss composability and reservation of resources.

We consider a tiled SoC that comprises a number of processor and memory tiles interconnected via a Network-on-Chip (NoC), as presented in Figure 2. A typical processor tile consists of a processor, a MicroBlaze core in our case, a set of local scratch-pad memory blocks, e.g., for instruction (IMEM), data (DMEM), and optionally hardware blocks to facilitate remote, outside tile data transfers, and to enable the overlap of communication and computation, e.g., Direct Memory Access (DMA) modules or communication assists [20]. The processor tile is further equipped with a timer to generate interrupts and implement the fixed duration user slots, as introduced below. A memory tile consists of a memory controller and a number of memory banks.

An application consists of a set of tasks, each of which executing sequentially on a processor. The tasks may be stat-

ically partitioned across multiple processor tiles to enable parallel processing. A light-weight Operating System (OS) executes on each core, provides applications with services, such as drivers to SoC resources, and schedules applications and their tasks. Our VM approach supports use-cases comprising best-effort and real-time applications, executing concurrently on the SoC platform. In what follows we proceed by presenting the models of these two application types.

The task forming a best-effort application communicate using distributed shared memory. Any programming model is supported, under the assumption that the applications must not employ any kind of resource locking of slaves shared between applications. A locked shared resource can be monopolised by an application, which affects the temporal behaviour of another application that attempts to access this resource, hence violates composability.

Tasks of real-time applications operate in a more restrictive fashion to ensure that their temporal behaviour can be bounded. As many applications in the firm real-time domain belong to the signal processing class, we choose a programming model that naturally fits the domain of streaming applications. In this model, each real-time task executes continuously, iteratively. Inter-task communication and synchronisation is implemented using logical FIFOs, with blocking read and write operations. This model enables overlapping computation with communication via a DMA engine, as presented below. It furthermore allows modelling an application as a data-flow graph, which enables efficient timing analysis. Note that best-effort applications may be implemented utilising a more relaxed version of this model in which the temporal behaviour of the tasks and the inter-task communication do not have to be bounded.

To reduce cost, many of the SoC resources may be shared. Composability requires strict reservations for each resource shared between applications [2]. For example, [11] proposes to realise a composable processor by Time Division Multiplexing the applications at the granularity of user time slots. A user time slot is a basic quantum of fixed duration that the OS allocates to an application. In between two user time slots, the OS schedules a new application task in a so-called OS time slot. The OS time slot should also have a fixed, application-independent duration to be composable. Resources used exclusively by a single application may have to be arbitrated between the application's tasks, but pose no problem to composability (no inter-application interference can occur). We denote such resources as not shared between applications, or shortly not shared.

3. COMPOSABLE VM CONCEPTS

The introduction briefly explains how a composable VM extends the scope of composability to independent application development, optimisation, compilation, and linking into an individual executable file, and enables dynamic loading of use-cases. Figure 3 illustrates the virtualisation range inside a processor tile. All processor's load/store operations and instructions fetch use virtual memory addresses that are translated by a Translation Look-aside Buffer (TLB). Misses in the TLB are served by a hardware page-table (PT). In this section we present this VM, our composable memory reservation, and the dynamic application set-up.

3.1 Composable MMU

The processor time is shared among concurrent applications, therefore conventional MMUs raise two problems with respect to composability. First, at the very start of a user slot, the content of the TLB is determined by the previous user slot, which can belong to the same application or to another one. Therefore, an unknown number of TLB misses will occur, which stall the execution for a number of cycles. Second, an application may swap-out pages of another application from the memory to the disk, causing future page faults to the second application. The TLB miss and page fault penalty depend on previously executed applications, therefore, the timing of the current application is no longer independent.

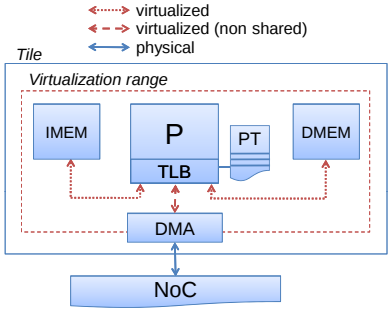


Figure 3: Memory virtualisation inside a tile.

To address the first problem, we assign each application a page-table, and, at any given moment, the page-table of the application running on the processor is entirely stored in a hardware table. The OS performs two actions before entering a user slot: (i) it invalidates the TLB entries, and (ii) it loads the page-table of the next application in the hardware table. The TLB still has a cache-like behaviour, but it is composable since cold misses will always occur after context switch, regardless of the application that executed before. Furthermore, the TLB misses are quickly resolved in hardware and do not raise an exception or interrupt; their penalty is hence significantly reduced when compared to a conventional VM.

The second problem does not exist in our system because we assume that each use-case fits in the local memory. This is a realistic assumption in embedded systems, which typically execute small applications. For performance reasons, each application is designed to have fast access to its private data and code, which are hence stored within the tile. As a result, we can ensure that all pages are permanently resident, i.e., pages are not swapped to off-tile memory during application execution. This avoids the large performance penalty (in the order of magnitude of millions of cycles) of page faults. The page management mostly involves allocation and de-allocation of physical pages and it is performed during OS boot and application set-up, i.e., use-case switch, thus during application execution no inter-application page swaps may occur. Furthermore, our approach may reduce the size of the page-table when compared to the more conventional approach of having one page table per task. The total local memory footprint of the tasks of an application that execute on a tile is smaller of equal than the physical local memory in that tile. Different tasks of the same application use different parts of a single range of virtual addresses, and a task needs hence only the page table entries to access its subset of this range.

Moreover, the address translation is predictable as the latency of a TLB hit is typically constant, i.e., 3 cycles for the utilised MicroBlaze core, and the maximum latency of a

TLB miss is bounded, i.e., it equals the latency of accessing the hardware page-table. Furthermore, page faults never occur, as mentioned before. Hence, the scheme is predictable, given that the TLB behaviour of a real-time application is predictable.

3.2 Composable Memory Reservation

This section presents the memory reservation mechanisms utilised for the OS services and for the user applications.

3.2.1 OS memory reservation

The drivers and OS code are denoted as the base code. The usage of OS services incurs overhead due to virtual memory. On one hand, without virtual memory, the location of these services is known at compile time and can therefore be statically linked. With virtual memory, on the other hand, such services are generally provided by system calls or dynamically linked libraries. In embedded systems both options are far too costly in terms of execution time. The first requires costly context switching and system call handling. The latter incurs dynamic overhead, such as symbol resolution and dynamic binding. Instead, we require an approach that not only incurs low overhead, but also has an application-independent latency, in order to be composable. The OS services are used during the application's time slot and thus the processor runs in virtual mode. Any overhead can be avoided if these services are called as statically linked functions. Our approach is to implement this by assigning these services a fixed position in the virtual address space as shown in Figure 4.

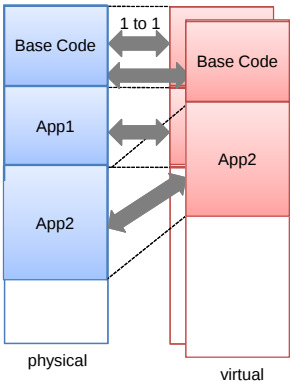
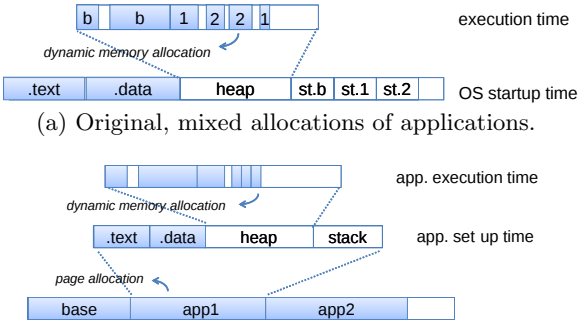


Figure 4: Memory reservation; the base code has a fixed position in memory.

3.2.2 User memory reservation

The user memory reservation follows a two-level strategy; the first level, inter-application, is *page allocation*, and the second level, intra-application, is *dynamic memory allocation*, as shown in Figure 5. As reference, the figure also shows the approach used in the original case where applications are compiled together. The two-level strategy is in line with the approaches used for reserving other composable resources as advocated by [11].

The pages required by an application are allocated at its set-up stage, at run-time, for both instructions and data (first level). The dynamically allocated memory space (heap) and the stack, however, can grow during application execution and therefore may require additional pages. In a composable system such a situation should not occur for two reasons. First, the time it takes for the OS to allocate a



(a) Original, mixed allocations of applications.

(b) Two-level approach, allocation per application.

Figure 5: Memory allocation approaches.

physical page depends on previous page allocations of other applications. Hence, the system would not be composable. Second, if applications can consume more memory than it was allocated at set-up, the memory request of one application can potentially not be satisfied because another application consumed all available memory. In that case, the system would also not be composable. To prevent these and ensure composability, page allocation is performed once, at application set-up. Thus applications request their worst-case needed memory size at once. This is not a strong limitation in embedded systems where resource-bound (worst-case) design is common practice.

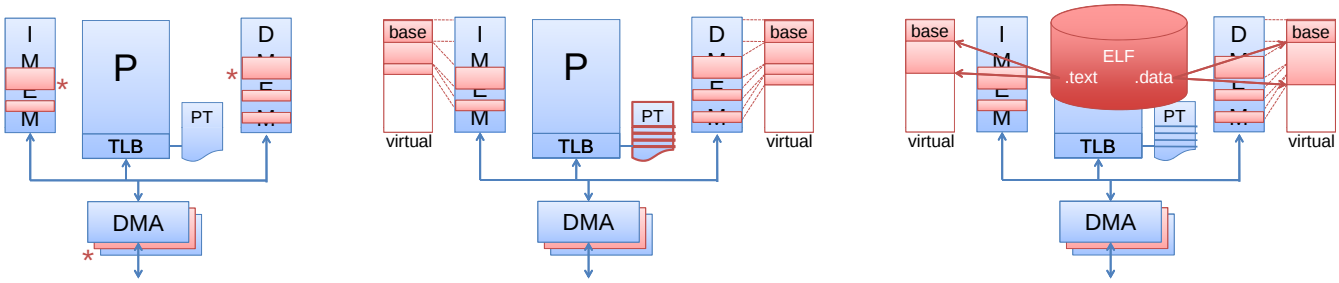
As a result of the composable page allocation, each application may use its own dynamic memory allocator (second level). The dynamic memory allocation during execution time (by a `malloc()`), is performed within, and bounded to, the allocated application pages. Therefore, there is no interaction between `malloc()` and the system memory management. The two are strictly separated, and as a consequence the latency of a call to `malloc()` is independent of any other application running on the system and therefore composability is maintained.

3.3 Dynamic Application Set-up

In this section we detail the steps involved in application set-up at run-time and its benefits. Figure 6 shows the process of setting-up an application using our virtual memory approach. In the first step the required resources, marked with '*' in Figure 6(a), are allocated. These resources include memory pages, and other tile resources, such as DMAs. In the instruction and data memory, the physical location of the allocated page is irrelevant. Moreover, the pages do not have to be in a single consecutive address range.

The second step of a set-up process (Figure 6(b)) is the virtual-to-physical address binding. The physical memory regions are bound onto the virtual memory pages. For each page an entry in the page-table is created. The drivers and OS code, i.e., the base code, are also bound into the virtual address space of the application, in a fixed position, as detailed in Section 3.2.1. The location of the application code in the virtual address space is determined at compile time and is not fixed. The location of each of the sections is stored in the executable (elf) header. This information is used to bind physical pages to the proper virtual addresses.

In the third step the application sections are actually loaded. Because all sections are placed at the virtual address determined by the compiler, all links to the base code are maintained. Moreover, the instruction and data sections do not have to be relocatable, hence are not restricted to rel-



(a) Allocation (b) Binding (c) Loading

Figure 6: The process of setting-up an application.

ative addressing.

Without virtual memory, all applications are compiled together and their data sections have a fixed position in the physical address range. With virtual memory and separate executables per application, dynamic binding is possible and local memory can be used more efficiently. When using virtual memory, the system can reuse local memory as it can dynamically allocate sections in the physical address space. This not only provides a benefit for the data memory, but also for the instruction memory.

4. COMPOSABLE VM ON MICROBLAZE

We demonstrate the proposed VM utilising Microblaze processor cores that already embed MMUs. This section describes the configuration of this MMU that implements a composable, predictable virtual memory.

The MicroBlaze core that we use in our platform provides a MMU with a two-level TLB, as presented in Figure 7. The Instruction and Data TLB (ITLB and DTLB), which can be configured to contain 1, 2, 4, or 8 entries are on the first level. The ITLB and DTLB are hardware managed. Any memory access goes either through the ITLB or the DTLB. If the corresponding page-table entry is found, the virtual address is translated into the physical address and the memory access is passed on to the bus interface. If the page-table entry is not found in the ITLB or DTLB, that entry is loaded from the second-level TLB, the Unified TLB (UTLB). The UTLB has 64 entries and it is software controlled. Its entries can be written by addressing a few special purpose registers.

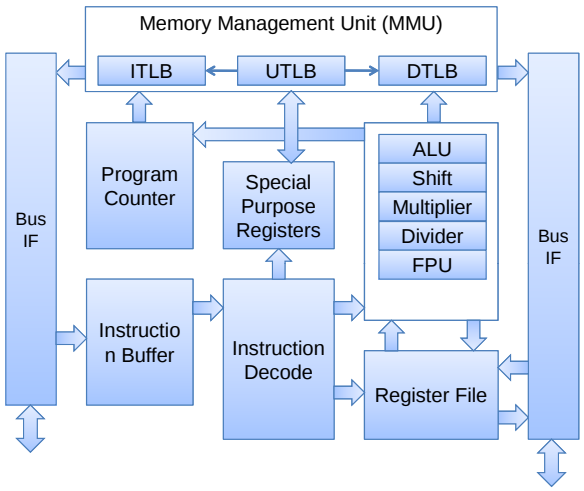
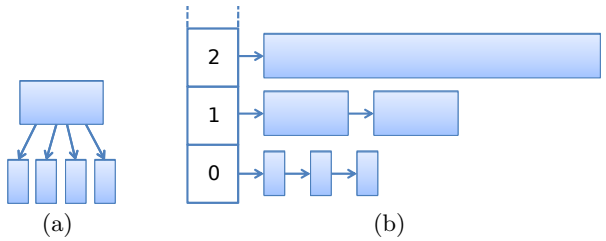


Figure 7: The MicroBlaze architecture.

As explained in Subsection 3.1, we propose to utilise a page-table that fully resides in the hardware, to achieve composability. The two-levels MicroBlaze TLB is configured such that the ITLB and DTLB to work as TLBs and the UTLB to work as the hardware page-table. The OS loads the UTLB and invalidates the ITLB and DTLB at every task switch. Page management is performed during OS boot and application set-up. Our implementation of page management is based on the Linux buddy system, as detailed in what follows.

The principle of the buddy system is that a larger page can be split into two smaller pages (buddies). Using the right combination of larger and smaller pages, a region of memory can be allocated which closely fits the requested size. Whenever a memory region is freed and the two buddies are both free, they can be combined to form the larger page again. The smallest page size is of order 0, the next of order 1, etc.

In the Linux buddy system, page sizes are powers of two. The MicroBlaze hardware only supports pages of powers of four, from 1KB upward to 16MB. To simplify the page management, our buddy system only supports these native page sizes. Therefore, in our implementation a page is split into four pages instead of two, as shown in Figure 8(a). All free pages are maintained in a linked list per order (see Figure 8(b)). Thus, to obtain a page of a certain size the corresponding list is accessed. The system starts with maximum sized pages. To obtain smaller pages, a larger page is split into four buddies, which are added to the proper list. A page that is allocated is removed from the list. When a page is freed it is added to the list of the correct order. If four buddies are found in the list, they are combined and the resulting page is added to the list of one order higher.



(a) (b)

Figure 8: The buddy system: (a) a page can be split in four smaller pages (buddies), (b) free pages are maintained in a linked list per order.

A potential trade-off exists between memory utilisation and TLB utilisation. Both memory and TLB are of limited size. The buddy system is designed to allocate the smallest memory region that fits the requested size, by using a

combination of pages. For example a region of 17KB would consist of a page of 16KB and one of 1KB, which is more memory efficient than allocating one page of 32KB. By using one additional page-table entry, 15KB of memory is saved. On the other hand, if a request for 15KB would be made, the buddy system returns six pages. Thus, five page-table entries are used to save one 1KB of memory. This trade-off is made at system integration time by specifying the memory reservation sizes that are passed on to the application loader program.

5. EVALUATION

The experimental platform consists of a CompSOC instance with two worker processor tiles, a monitor tile, and a memory tile, all communicating via an on-chip interconnect. The monitor tile gathers information about the execution on the worker tiles and sends it to a host PC. Each processor tile and the monitor tile include a Microblaze core with separate instruction, data, and communication memories. DMA modules are responsible for remote, outside-tile data transfers. This platform is implemented on a Virtex 6 FPGA; all the resources run at clock frequency of 50 MHz. Note that we utilise a cycle accurate FPGA prototype of the entire SOC, hence our results are more realistic than most existing SOC simulators.

Each MicroBlaze executes a composable OS similar to the one in [11]. The workload consists of a JPEG decoder, and a simple synthetic application ($A1$). Each of these two applications consists of a set of communicating tasks. The JPEG include three tasks: a variable-length decoder (vld), an idct, and a colour conversion (cc). The vld is mapped on the one tile and the idct and cc are mapped on the other tile. The synthetic application consists of five tasks, communicating data from one-another, in a pipeline. The first, third and fifth tasks are mapped on the same tile as the vld, and the second and forth tasks are mapped on the other tile with the idct and cc. In the rest of this section we present experimental results indicating VM composability and we discuss the costs involved in implementing it.

5.1 Composability

The entire platform is carefully designed to be composable by construction, meaning than applications do not interfere, even with one clock cycle. To experimentally illustrate the composability of our VM approach we first run JPEG alone [J] and then we run JPEG with a synthetic application concurrently [JS]. We measure the execution time of all three JPEG tasks in both cases using a hardware timer. We also register the exact start and stop times of the tasks. Comparison of the two traces indicates no difference for the JPEG application when running alone and with the synthetic application. To illustrate this, Figure 9 shows, for each vld task iteration, the difference between the two use-cases [J] and [JS], which is a flat line at value 0. For comparison reasons, we also include a line that shows non-composable behaviour. It compares the same two use-cases, with a non-composable MMU, i.e., using a single page-table for all applications stored in the UTLB. Furthermore, although not shown in this paper due to lack of space, the start and end times of the tasks are completely equal. For the other two tasks of JPEG we obtain the same results. In summary we can conclude that the synthetic application creates no functional or temporal interference on the execution of JPEG.

We claim that the order in which applications are loaded

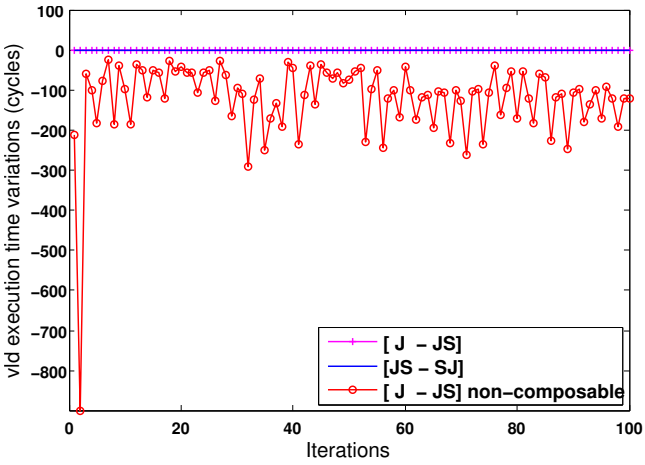


Figure 9: Difference in execution time of the vld task of JPEG for different use-cases.

does not influence composability. To prove that, we compared two use-cases those both consists of JPEG and a synthetic application, but have a different loading order, [JS] and [SJ], respectively. Figure 9 also compares these two use-cases. The difference is again a flat line at value 0. From these results we conclude that composability is obtained.

5.2 Cost Analysis and Limitations

First, we analyse the area cost caused by introducing virtual memory. The two Microblaze worker cores have been extended with an MMU, which requires FPGA resources. From the synthesis report we derived the values of Table 1. The first row contains the resource utilisation of a single Microblaze core, consisting of the components as indicated in Figure 7. The table shows that the Microblaze core requires about 80% more flip flops and LUTs, which is a significant increase. The Microblaze is a very small and simple core compared to modern embedded processors, which explains the large relative increase in size. In absolute terms, the increase is of normal proportion. This is confirmed by the fact the for the total SoC the increase in utilisation of flip flops and LUTs is only 3.2% and 5.3%, respectively.

Considering the current platform, the BRAM utilisation is most relevant as BRAM blocks are the scarcest resource, limiting the number of tiles we can implement. The table shows that an MMU requires 1 BRAM block. By equipping two tiles with virtual memory, the BRAM utilisation of the SoC increases by 0.7% only.

Second, we discuss and analyse the performance impact due to virtual memory. In all VM implementations, composable or not, the address should be translated by the TLB, which incurs an extra delay that affects each load/store and the instruction fetch. For example, the local memory latency on the MicroBlaze increases from 1 to 3 cycles when VM is enabled. Furthermore, at task switch and OS service calls the address space is also switched, which may involve an overhead. All these overheads are determined by the MMU design and should be as small as possible.

To assess the impact of this increased memory access latency on the performance of applications, we measure the execution time of the first 100 JPEG task iterations. We used very large task slots to avoid task preemption from altering the results. Figure 10 shows the cumulative execution

Table 1: Increase in FPGA resource utilisation due to MMU hardware.

	Flip Flops			LUTs			BRAMs		
	Original	VM	Increase	Original	VM	Increase	Original	VM	Increase
Microblaze Core	1304	2338	1034 79.3%	1536	2748	1212 78.9%	0	1	1
Total SoC	65224	67292	2068 3.2%	45455	47879	2424 5.3%	270	272	2 0.7%

time for both the original case and for using virtual memory. The total execution time of these 100 iterations increased by 39%. Misses in the ITLB and DTLB are hardly of influence on this number. Both contain 8 entries, which is sufficient for JPEG. Therefore, only at the start of the application slot a few misses occur.

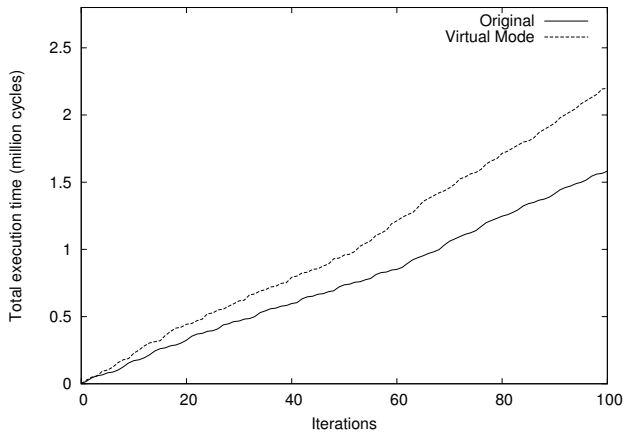


Figure 10: The cumulative execution time of the first 100 task iterations of JPEG.

In addition to this unavoidable VM overhead, in our system the hardware page-table is updated and the TLB is cleared at each task switch. However, the page-table update takes at most a few hundred cycles, and the TLB misses penalty is small. On our MicroBlaze core, the task switch would cause a maximum 8 misses, each of which taking 32 cycles to be resolved. Moreover, task switches do not occur frequently. The OS execution takes approximatively a thousand cycles, and a user slot is at least 50 thousand cycles, which, represents one milliseconds at a clock frequency of 50MHz. This time values ensure a proper response time for a typical real-time OS. Hence, the VM-related task switch overhead is under 1% and it can thus be considered minor.

Other two sources of large performance penalty in conventional MMUs are page faults and TLB misses that may cause events or interrupts that have to be treated by the OS. The ITLB and DTLB miss penalty on the MicroBlaze is maximum 32 cycles, and it is resolved in hardware, hence costly events or interrupts do not occur. Furthermore, the composable VM has no page faults. Therefore the penalty due to these two sources is much smaller than the one in a conventional MMU.

Moreover, we measure the increase of the application set-up time due to VM. Our experiments indicate that the set-up time of JPEG increase with 5.3%. This set-up time, however, depends on the state of the buddy system and the requested size. The best-case condition is when the requested region is a single page, a page of the order is available, and the order is the maximum value. Artificially creating this best-case condition, the set-up time of JPEG increased by 3.0% compared to running without virtual memory. The worst-case condition is when the memory is completely fragmented into the smallest possible pages, case in which the set-up time increases by 44.7%.

Third, another cost associated with using virtual memory is the increased size of the OS and thus its memory utilisation. The elf file of the OS increased from 15KB to 23KB, which is mostly increase in instruction memory utilisation. Further, memory management uses some additional heap space, but this depends on the number of applications that are running. Relatively, the OS size increase is significant but one has to consider that the original OS is extremely light-weight. In absolute terms, the elf file increased 8KB which is modest.

Finally, the limited page-table size is not a major restriction because, as our approach is tile based, only the memory mapped resources inside the tile have to be translated. As these are limited in size (typical local memory sizes vary from 16 KB to 256 KB) each application requires only a few translation entries. Including the entries for the base code, applications typically require 10 to 20 page-table entries.

6. RELATED WORK

The related work falls into three categories, namely composable SoCs, memory management for dynamic use-case switching, and virtual memory for embedded systems.

Predictability is a conventional requirement for embedded systems, and it is realised in many multi-core SoC platforms [3, 20, 19, 15, 22]. Next to predictability, recently composability has been advocated [16, 23, 2] and demonstrated for SoC resources: memory controllers [1], network-on-chip [9], operating system [11], and entire platforms [17, 2]. Moreover, for an interconnect multiple undisrupted use cases were demonstrated in [10].

To minimise the memory usage when applications have highly variable memory demands or when the use-case changes, existing approaches typically take two steps. First, the application is thoroughly analysed and the memory hierarchy is synthesised at design-time [13]. Second, data are explicitly copied from a memory block to the other, when reconfiguration is necessary, at run-time [6, 8, 24]. The main restrictions of these approaches are that (i) the application code needs to be modified, i.e., API calls to manage data copying have to be inserted, and (ii) applications should consist of a set of affine indexed loops. These are not strong limitation for hard/firm real-time applications which have to be highly analysable, but they are not common in soft real-time and best-effort applications. In contrast, virtual memory enables execution of already compiled and linked applications, with no restrictions on the application code.

Several approaches to virtual memory for real-time embedded systems exist in the literature. Some have proposed novel MMU architectures or page allocation strategies and mainly target improved predictability and energy efficiency [25, 18, 14, 12]. Other works include the software management part of virtual memory. The approach in [5] provides predictable VM for safety-critical systems, by only

allowing best-effort applications to use the MMU. In [21] a memory management method, including an MMU architecture, is proposed that provides deterministic allocation of global memory on a SoC. In [4] the scratch-pad memory is virtualised, and the applications may call VM API to create, allocate, swap, or destroy virtual pages. For systems that lack an MMU, in [7] a software VM approach is proposed.

While all these approaches provide interesting ideas in the field, to the best of our knowledge, we are the first to propose a composable and predictable virtual memory scheme.

7. CONCLUSIONS

In this paper we introduced a composable, predictable virtual local memory scheme for a tiled multi-core SoC executing multiple applications. As a result each application can be independently developed, coded, and linked into an individual executable file, and loaded on the SoC at run-time. Run-time application loading reduces the required memory footprint from a worst-case use-case containing all possible applications that run on the processor, to only those that actually run concurrently. A two-level memory reservation scheme allocates static, per-application virtual-memory budgets, and allows each application to perform its own dynamic memory allocation, within its budget. At run-time the virtual addresses are dynamically translated to physical addresses in a composable and predictable way.

We implemented this virtual memory on a MicroBlaze core with a built-in Memory Management Unit. We experimentally demonstrated composability in an SoC modelled in FPGA, comprising two MicroBlaze cores and executing an JPEG decoder and a synthetic application. On this platform we also found that the area required by the default MMU is relatively small (3.2%, 5.3%, and 0.7% of the total FPGA flip flops, LUTs and BRAM blocks, respectively), the operating system code required to program the MMU increased with 8 KB. Furthermore, we found that VM in general is expensive; using the MicroBlaze's built-in MMU incurs a performance loss around 39% due to address translation latency. In addition, the implementation of composability on this VM avoids the large penalty of page faults, minimises the penalty of TLB misses and has a negligible overhead, below 1%, due to loading the page-table and invalidating the TLBs at each application switch.

8. ACKNOWLEDGMENTS

This work was partially funded by EU projects Catrene CA104 Cobra and CA501 COMCAS.

9. REFERENCES

[1] B. Akesson et al. Composable Resource Sharing Based on Latency-Rate Servers. In *DSD*, 2009.

[2] B. Akesson et al. Composability and Predictability for Independent Application Development, Verification, and Execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, 2010.

[3] A. Andrei et al. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*, 2008.

[4] L. Bathen et al. SPMVisor: dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. In *CODES*, 2011.

[5] M. D. Bennett and N. C. Audsley. Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems. In *ECRTS*, 2001.

[6] D. Cho et al. Adaptive scratch pad memory management for dynamic behavior of multimedia applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2009.

[7] S. Choudhuri and T. Givargis. Software Virtual Memory Management for MMU-less Embedded Systems. Technical report, University of California, Irvine, 2005.

[8] P. Francesco et al. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, 2004.

[9] K. Goossens and A. Hansson. The Aethereal Network on Chip after Ten Years: Goals, Evolution, Lessons, and Future. In *DAC*, 2010.

[10] A. Hansson et al. Undisrupted Quality-Of-Service during Reconfiguration of Multiple Applications in Networks on Chip. In *DATE*, 2007.

[11] A. Hansson et al. Design and Implementation of an Operating System for Composable Processor Sharing. *MICPRO*, 2011.

[12] D. Hardy and I. Puaut. Predictable code and data paging for real time systems. In *ECRTS*, 2008.

[13] I. Issenin et al. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC*, 2006.

[14] M. Kandemir et al. Compiler-Directed Code Restructuring for Reducing Data TLB Energy. In *CODES*, 2004.

[15] P. Kollig et al. Heterogeneous multi-core platform for consumer multimedia applications. In *DATE*, 2009.

[16] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[17] H. Kopetz and G. Bauer. The Time-Triggered Architecture. Number 1. IEEE, 2003.

[18] J. Lee et al. A Banked-Promotion TLB for High Performance and Low Power. In *ICCD*, 2001.

[19] O. Moreira et al. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT*, 2007.

[20] A. Shabbir et al. CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications. *J. Syst. Archit.*, 2010.

[21] M. Shalan and V. Mooney. A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip. In *CASES*, 2000.

[22] M. Wiggers et al. Modeling and analyzing real-time multiprocessor systems. In *CODES*, 2010.

[23] R. Wilhelm et al. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Trans. on CAD*, 2009.

[24] C. Ykman-Couvreur et al. Design-time application mapping and platform exploration for MP-SoC customised run-time management. *IET Computers & Digital Techniques*, 2007.

[25] X. Zhou and P. Petrov. The Interval Page Table: Virtual Memory Support in Real-Time and Memory-Constrained Embedded Systems. In *ICSD*, 2007.