PRBE

# Abstract

* Is 8-bit sufficient for a subset of embedded applications, and can intelligent memory design help?
* Memory Safety and Virtualisation will empower software developers and allow for multi-programming on enabled devices.
* Use of picoBlaze as a compact 8-bit softcore processor. Design Extensions to allow for hardware support of operating system functions
* Due to its small size, can we add multiple pb cores to one chip and see improved performance(?) over a single, more powerful(?) core. Is this worth the increase in design complexity, power, silicon space and cost?
* How much OS functionality can we implement as hardware, allowing maximum memory utilisation for application software

An intelligently-designed memory system is integral to creating the Virtual Machine abstraction provided by multiprogrammed operating systems, as well as maximising the performance of hosted applications. In spite of advances in CPU architecture and memory design, the greatest limiter to system performance is the diversity of software applications and the high level of unpredictability in how these applications access instructions and data in memory.

By focussing on the embedded software space, *specifically (Real Time Operating Systems?),* we can reduce the set of applications that will need to be run on a given system and hence design a device that is better specified to run these types of application with much improved performance.

In this report we describe the construction of a System-on-Chip that uses a multicore architecture with shared memory virtualisation implemented on the Spartan3E 4000 FPGA. The foundation for each core is the picoBlaze softcore 8-bit processor. This paper will concern the performance of memory accesses distributed across multiple cores in an embedded context.  
The paper discovers that Sam’s design is amazing, increases bandwidth to 4Tbits/s, reduces latency to 67picoseconds per block retrieval and magically transmutes any application code into an innately parallelised, superpipeline optimised stream of wonder and computational gorgeousness.

# Introduction

# Literature Review

* Mooney & Shalan – A Dynamic Memory Management Unit for an Embedded Real Time SoC
* Meenderinck & Goossens – Composable Virtual Memory for an Embedded SoC
* Whitham & Audsley – The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing and Case Study
* U. Drepper – What every programmer should know about memory
* Spartan 3E User Manual

Dynamic Memory Allocators

The responsibility of providing extra memory to a process at runtime (dynamic memory allocation) is delegated between the memory management subsystem exposed by the kernel, and a user-space allocator provided by a library or language run-time. The logical region within a process’ address space which is reserved for dynamic allocations is called the heap, and its size and location within the address space is handled by the kernel. Whenever a request is made by the process for extra memory, the kernel will expand the size of the heap and initiate whatever subsystems are necessary (i.e. the MMU **NB what else**)for the device to assign a new page frame to the process’ address space, ready for the process to use.  
The memory allocator is responsible for managing the memory requirements within the process, keeping track of present allocations, free space and fragmentation (i.e. maximising the utilisation of the heap and reducing unused space between allocated blocks). The exact implementation of an allocator should strive to make memory management as efficient and deterministic as possible, and should be tailored to the patterns of memory usage required by the platform. At an application level, the allocator should be leveraged to ensure a process cooperates well with other memory users, if not, it is the kernel’s duty to restrict or kill such a wayward process.