apple core

SEVENTH FRAMEWORK PROGRAMME

FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Report on Porting Operating System to SVP/Microgrid Platform

Deliverable D5.3, Issue 1.0

Workpackage WP5

Author(s):	M.A. Hicks, R. Poss, C. Jesshope, M.W. van Tol, M. Lankamp		
Reviewer(s):	M.A. Hicks, R. Poss, C. Jesshope, M.W. van Tol, M. Lankamp		
WP/Task No.:	WP5	**Number of pages:**	81
Issue date:	2010-09-30	**Dissemination level:**	Public

Purpose: The purpose of this deliverable is to describe the work carried out, and progress made, on the porting of operating system support to the SVP/Microgrid platform.
Results: The main results of this deliverable are to be found in the investigation and implementation in four key areas: resource management/scheduling, where a fully-fledged resource allocator is implemented; I/O, where both a novel operating system stack and Microgrid hardware I/O Cores are implemented (in μTC and emulation); a deadlock prevention scheme, implemented as an adjunct to the SL toolchain; monitoring support, to quantify over time both the behaviour of individual programs and their impact on the overall architecture; and additional application library support.
Conclusion: Key issues have been identified and resolved in the following areas: architecture support for scalable I/O; space allocation of cores for software components; space allocation of memory for heap and stack management; concurrency monitoring; library compatibility for legacy program code. In particular for I/O, a bespoke, decentralised operating stack is implemented in μTC and special, reduced complexity, I/O cores are implemented in the Microgrid to facilitate highly parallel and scalable device I/O. This is provided via a standard interface to client applications. Also, the issue of deadlock prevention which was raised during the initial phases of the project has been resolved through a co-design of a hardware-directed software prevention scheme.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2010-09-30

Document history

When	Who	Comments
2010/08/24	M.A. Hicks	Initial Outline
2010/08/31	M.A. Hicks	Added Initial Draft on I/O
2010/09/03	R.C Poss	Added Initial Draft on Monitoring
2010/09/03	R.C Poss	Added Initial Draft on Resource Allocation
2010/09/06	C.R. Jesshope (via M.A. Hicks)	Added Initial Draft Overview
2010/09/07	M. Lankamp	Added Initial Draft on Deadlock Prevention
2010/09/07	M.A. Hicks	Added Notes SL5,7,8 & SVP21,31 as Appendices
2010/09/08	M.A. Hicks	Added 'at-a-glance' Section to Overview
2010/09/10	M.A. Hicks	Initial First Full Draft
2010/09/15	All Authors	Feedback from First-draft
2010/09/29	All Authors	Final Version

Project co-funded by the European Commission within the 7^{th} Framework Programme (2007-11).

Table of Contents

1 Overview **1**
- 1.1 Introduction and Motivation . . . 1
- 1.2 An OS strategy . . . 2
  - 1.2.1 Space vs Time . . . 3
  - 1.2.2 Processes vs. Threads . . . 3
  - 1.2.3 To Preempt or not to Preempt . . . 4
- 1.3 Deliverable 5.3 'at-a-glance' . . . 5

2 Resource management **6**
- 2.1 Spacial on-chip resource management . . . 6
- 2.2 Thread and family context management . . . 7
  - 2.2.1 Thread local storage . . . 7
- 2.3 Memory storage . . . 8
- 2.4 I/O Channels and other devices . . . 8
- 2.5 Summary . . . 8

3 Monitoring **9**
- 3.1 Introduction . . . 9
- 3.2 Synchronous in-program monitoring . . . 9
  - 3.2.1 Low-level support for performance counters . . . 10
  - 3.2.2 Software-hardware interface . . . 12
- 3.3 Asynchronous architectural monitoring . . . 12
- 3.4 Summary . . . 14

4 Input/Output in SVP and Microgrids **15**
- 4.1 Overview . . . 15
  - 4.1.1 Motivation . . . 15
  - 4.1.2 Context of I/O Work . . . 15
  - 4.1.3 Related Work . . . 15
  - 4.1.4 Key Areas of the Work . . . 16
- 4.2 I/O Operating System Stack . . . 17
  - 4.2.1 μTC I/O API . . . 17
  - 4.2.2 I/O Model (Synchronous and Asynchronous) . . . 18
  - 4.2.3 Low-Level Drivers . . . 20
  - 4.2.4 I/O Places . . . 20
  - 4.2.5 Parallel I/O . . . 20
- 4.3 Microgrid I/O Implementation . . . 20
  - 4.3.1 I/O Cores . . . 21
  - 4.3.2 High-Speed Bus . . . 21
  - 4.3.3 Bus Interface . . . 22
  - 4.3.4 Device Communication and Interrupt Handling . . . 22
  - 4.3.5 Synchronising and Interrupts . . . 23
  - 4.3.6 Memory Interface . . . 23
- 4.4 Summary . . . 23
  - 4.4.1 Performance Results . . . 24
  - 4.4.2 Milestones . . . 27
  - 4.4.3 Future Work . . . 27

5 Cooperative Deadlock-Prevention **28**
5.1 Problem Description . . . 28
5.2 Sequentializing . . . 28
5.3 Registers . . . 29
5.4 Group creates . . . 29
5.5 Delegated creates . . . 29
5.6 Exclusive creates . . . 29
5.7 Hardware extensions . . . 30
5.8 Summary . . . 30

6 Report Summary **31**

Appendices **33**

A SL Library: dynamic place allocation (TR) **34**

B Asynchronous simulation monitoring (TR) **39**

C SL Library: performance counters (TR) **46**

D Generalized I/O events for the Microgrid (TR) **52**

E Towards a Microgrid Hardware I/O Mechanism (TR) **58**

F Efficient heap allocation on shared memory SVP (TR) **63**

G Simple Example SL Program Utilising I/O API **74**

H SL Standard Library (TR) **75**

1 Overview

1.1 Introduction and Motivation

There is still hesitancy in the semiconductor industry in embracing large scale concurrency in microprocessor design. This is especially true in mainstream computing, where we still have processors with a relatively small number of complex cores. However, in other domains, e.g. gaming and graphics engine, there is a clear benefit shown from the use of a larger numbers of simpler cores. The clear distinction in usage is that the former comprises a complete system that must provide a whole range of system services whereas the latter are accelerators that execute a single task at a time relying on a host to provide whatever system services are required. In Apple-CORE we view these as two extremes of use and our operating system development aims to bridge this divide with a coherent strategy. What is required is a strategy that embraces execution model kernel, resource management, security and other system services that make up a commodity system. Coupled with this, there is a growing motivation for energy-efficient computation, which is one of the key advantages of going to larger numbers of simpler cores. This report outlines the issues in making many-core chips into general purpose computing devices and reports on the developments that have been achieved in the Apple-CORE project. In the original project DoW we had anticipated a complete port of a Unix-like kernel onto the SVP core. This was later modified in [3] to adopt a progressive strategy that would rely initially on a host system for system services, which would gradually be migrated onto the Microgrid itself.

It should be emphasised that the strategy we have developed in Apple-CORE is applicable to a wide variety of many-core processor designs in addition to the Microgrid. In the future, we are likely to see general-purpose processors comprising much larger numbers of cores, this is inevitable. By the end of silicon scaling we could certainly see 1,000s perhaps even 100,000s of cores in a system. These core are likely to be heterogeneous, i.e. fat cores, clusters of thin cores of various sizes, integer cores, floating point cores, special purpose cores, reconfigurable hardware, etc., each with different time scales for managing concurrency, i.e. concurrency creation latency, computational throughput, synchronisation latency, communication throughput, etc. These systems will face the same problems of resource contention and allocation and the provision of reliable and secure services in a distributed environment.

With so many cores and in a general purpose environment there will be too much dynamism to map computations statically, the mapping process for a given task will have to compete with existing jobs that have already acquired resources but we still need map any new task to the most appropriate type of resource in order to meet any requirements such as latency, frequency or throughput as well keep within (physically constrained) power budgets. Given a mapping onto a set of resources, we expect code to have too much parallelism to schedule manually. This means that units of work must be scheduled fairly and efficiently in such a way that, for any set of resources, we can guarantee freedom from deadlock. We already have this property in SVP given sufficient resources (i.e. freedom from communication deadlock) and this property needs to be extended to guarantee freedom from deadlock where unbounded concurrency (with dependencies) is mapped to a finite set of resources.

If mapping to resources is dynamic, then for heterogeneous resources, code can not be specialised manually. Automatic specialisation is required for granularity control in order to amortise concurrency management latencies and also for managing deadlock. This will involve collapsing some of the concurrency exposed in order to meet the constraints imposed by the selected resources or may even require binary code transformation, e.g. mapping from a regular ISA to a function implemented in logic. In the SVP core, the same binary code can run in a singe thread slot on a single core or may be distributed to many cores each providing many threads. The core automatically supports the necessary reduction in concurrency in breadth (i.e. threads within a family) but compiler support is also needed for depth of concurrency in our hierarchical model. Again, we do not see these techniques as being limited to cores where the SVP computational model is built into the implementation of the ISA. In implementations of SVP on different cores the concurrency overheads

are likely to be larger and new techniques required for code transformation. Complimentary work is already being undertaken to implement efficient software SVP kernels and to provide a framework for code specialisation [24].

1.2 An OS strategy

To start this section we review recent activity in both industry and the research community. In March this year, Dave Probert, a Microsoft kernel engineer, gave a talk at the University of Illinois (`http://www.upcrc.illinois.edu/seminars.html`) entitled Beyond Kernels: On the Future of Operating Systems. In this talk he spoke about the desire for responsiveness and questioned the way current operating systems shares processor cores between multiple applications. He suggested that with many cores, it makes more sense for cores to be be dedicated to particular processes, with the OS acting more as a hypervisor, i.e. assigning work to cores an allowing the cores to get on with it. He also said that this approach was not supported across the board in his team. This approach, however, is at the core of the Apple-CORE strategy.

As another data point in this trend, Apple recently introduced the technology *Grand Central Dispatch*[1] in their flag operating system Mac OS X. This implements task parallelism based on transparent (system-managed) thread pooling. Using this framework, developers are invited to describe fine grained, asynchronous units of work in a way independent of hardware properties; the system then automatically dispatches tasks over a multicore execution engine. In order to achieve this, Apple introduced language extensions to their C/C++/ObjC tool chain that simplify the expression of concurrency and push programmers to isolate asynchrony and dependencies in their code. The Apple-CORE strategy is similar to GCD in that it introduces language extensions as a vessel to express target-neutral concurrency, while the environment takes responsibility for spreading tasks to hardware automatically.

In general, this approach is more common in the grid community, where it is common for jobs to be allocated sets of processors for a given task; it is also common for each processor resources to come with a complete time sharing operating system. The ExtremeOS European projects is developing Linux to support such grid applications. "The overall objective of the XtreemOS project is the design, implementation, evaluation and distribution of an open source Grid operating system? The approach we propose is to build a Grid OS based on the existing general purpose OS Linux." In the grid world jobs are large and may run for days, the emphasis in this project therefore is more on the support for virtual environments rather than the efficient execution of small jobs [2].

Another approach to the development of operating systems is the Barrelfish project[2], which is a joint project between ETH Zurich and Microsoft Research Cambridge. An interesting recent paper from this project looks at the design principles in creating an operating system for future many-core heterogeneous systems [1]. In particular it looks at scheduling. In their strategy they conclude that time multiplexing of cores will still be necessary, implying that resource management cannot solely operate by partitioning the resources of a machine at the point when a new application starts, which in their setting means operating system intervention. However, they do not consider jobs as being highly threaded and able to self-schedule themselves on available resources. Another interesting observation is that scheduling must be managed at multiple time granularities, which implies identifying units of work at different granularities.

The holistic approach adopted in Apple-CORE takes all of these issues on-board but does so without introducing complex scheduling algorithms. Indeed this would be counter productive, as with hardware support for the kernel, we have shown it is possible to allocate a cluster of 64 cores, create and execute 4K threads (Livermore Kernel 7) on them, synchronise on the completion and to do all of this within less than 4,000 cycles. Moreover, the average utilisation on all cores during this process is a 60% usage of all pipeline cycles. This does not allow much time for executing a scheduling algorithm.

In the remainder of this section we outline the strategy for the Apple-CORE approach

[1] `http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf`

[2] `http://www.barrelfish.org/`

1.2.1 Space vs Time

One of the key issues discussed in the background above is the issue of scheduling jobs or tasks. A strategy for scheduling is dependent on the availability of different resources in the system. If we look far enough back we come to an era where both compute cycles and memory were in short supply. This was the era of batch processing system, where one, in some cases a few jobs, were scheduled manually onto the single CPU or core. Computer systems for the last few decades on the other hand can be characterised by a surfeit of memory but but where compute cycles have still been a limiting factor, this led to the time sharing operating systems becoming widespread. Here, large numbers of jobs reside in memory and compete for CPU cycles. The progression from cooperative to preemptive scheduling allowed the prioritisation between different jobs to be managed by the operating system rather than collaboratively. Again the landscape has changed (is changing) and we are moving into an era where there will be a surfeit of both memory and CPU cycles. The constraints are going to be in how we use those cycles as, in these many-core systems, energy and communication bandwidth are going to be limiting factors. In this respect, the time sharing OS is no longer fit for purpose, as it is not energy efficient. It requires moving state into and out of the CPU as contexts are switched, consuming both energy and bandwidth unnecessarily.

In SVP we have a single composition mechanism from fine-grain threads to complete systems. The key support for the OS kernel in this composition is the ability to schedule a unit of work to specific place (a guaranteed set of resources) at any level of the composition, giving very efficient mechanism to implement space sharing. The strategy we adopt in SVP is to allocate resources for exclusive use of a thread, hence the ability to guarantee resources. This may be constrained by contract in energy and/or duration. The important issue is that there is a static guarantee of the resources allocated, which may be used at compilation time, e.g. to statically guarantee freedom from resource deadlock. In the case of the Microgrid we allocate one or more cores in a cluster. Each core provides a fixed number of families and threads and depending on the code's requirement for registers a fixed number of synchronising contexts.

On a single core we must still implement time sharing between the threads and this is implemented at a very fine grain in hardware. No thread may execute for more than one cache line's worth of instructions before yielding to another thread. The time slice is thus from 1 to 16 ns for a 1GHz clock and slices are allocated using the simplest round-robin algorithm amongst all threads that are able to proceed. This scheduling is starvation free but is considered undesirable if the sizes of the jobs or tasks vary significantly. However, in our strategy we will manage job size through resource partitioning and prioritisation by the size of partition allocated to a job. Thus, we see no reason to prioritise threads on a single core.

There are some exceptions to the strategy of allocating complete cores to jobs. For example legacy code and high level control threads in an application may be allocated exclusive use of a single thread on a core. However, the exclusivity of use for that resource can only be guaranteed if the thread does not create any further threads (legacy code) or only creates threads on other resources that have been allocated (control thread). It does not matter so much if a processor allocated a number of control threads is idle for any length of time, so long as it is not consuming power. It is a repository of synchronising state. With SVP implemented in the ISA of the core, we have a kernel that implements time-sharing between a fixed number of threads with a time slice of a few nanoseconds and a delegation mechanism that can schedule work to remote resources efficiently with time slices from a few microseconds upwards. For comparison, in Linux, a time-slice is between 5-100ms, depending on its niceness. A very nice process (19) gets just 5ms of CPU time before being interrupted. Therefore the microgrid can time-slice some six orders of magnitude more responsively than Linux and even space-slice three orders of magnitude more responsively than Linux can time-slice.

1.2.2 Processes vs. Threads

We have already seen that in SVP we have a common composition mechanism from threads to complete jobs, they are all SVP threads. Because of this we need a strategy to manage issues that

normal operating systems deal with by a partition of the units of work, namely between processes and threads. Perhaps the major issue is in memory protection. Most operating systems combine memory mapping and protection by the use of this partition. Processes are heavyweight units of work that have their own unique mapping from virtual to physical address, whereas threads are lightweight units of work that exist within a process. It is because mapping and protection are intimately connected that we have this partition, limiting protection to coarse-grain units of work. The result being that protection is coarse grained. The solution is to separate these concerns of mapping and protection, which allows protection at any level of granularity. This would support the SVP model where there is a common composition mechanism across granularities. There are many advantages to this approach as outlined in [32]. The major advantage is in optimising system services by avoiding the copying of data between address spaces [19], for example between OS and user spaces in issues such as networking and I/O. In the Microgrid our strategy is to use a single address space on chip and to overlay this with a fine grain memory protection based on protection domains. Prior research that has influenced our strategy has been undertaken in both coarse grain single-address space operating systems (SASOS) such as Mungi [11] and in fine-grain conventional systems such as Mondrian [32]. Our work in this area has already been reported in an earlier deliverable [21].

1.2.3 To Preempt or not to Preempt

A final plank in the strategy for an SVP operating system is to answer the question whether we need support for preemption. There are three main reasons for using preemption in an operating system and we have a strategy to deal with each of these. The first are hardware exceptions such as arithmetic anomalies and protection violations. For these we need to support an exception mechanism in the processor. Prior research was undertaken in the design of an exception mechanism for the SVP core [30] and in the current software emulation it does not make sense to implement this, as it would only slow down the processing rate of the emulator and can be managed by the underlying system. Of the solutions outlined in [30] our preference is for the simplest solution, which reserves a single thread context for executing an exception. During exception processing, which is not the common case, we accept a relatively inefficient single thread execution mode. This is the only form of preemption required in the core. On encountering an exception, the pipeline is flushed and switches to exception processing mode. Normal execution continues when the exception has been handled.

The second reason for normally requiring preemption is to manage signals between different parts of the system. This occurs when synchronisation and/or communication is required between unrelated parts of the system [23]. I/O is a classical example. In the SVP core we have a built-in mechanism for signalling, which is the synchronisation on writing a register or on family termination (which writes a register in the parent thread) and we have proposed both asynchronous and synchronous methods to use this at a system level, i.e. between unrelated concurrency trees [30, 23], which have been used to build I/O into the Microgrid [22, 12, 13].

The final reason for preemption is for managing scheduling, where a timer interrupt can allocate processor cycles in different quanta according to priority and niceness. As described in subsubsection 1.2.1, we have no need for preemption within a single core due to the fine-grain time-slice built into the hardware. There could however be a requirement for preemption in space-sharing resources. Without any form of preemption, we rely of cooperation in order to relinquish resources. A strategy to move away from this reliance on cooperation lies in the resource allocation itself. When a resource is allocated, it is possible to make a contract for the use of that resource, with a given time or energy budget. Then when that contract has expired the system can preempt the resources using one of two mechanisms. The softest is to reset the capability the thread has to create threads on that resource and wait for the job to terminate or alternatively the system can kill all threads at a place and reset it for further use.

1.3 Deliverable 5.3 'at-a-glance'

Table 1: Deliverable 'Checklist' (at month 33)

Milestone	Details	Due	Status	Addressed in...
D5.3	Port of operating system onto emulation platform	m33	∞	§1.2
M5.3.1	Investigate and chose microkernel base for OS port	m18	✳	§1.2
► **N1**	Resource Management	m33	☑	§2; App. A & F
► **N2**	Monitoring	m33	☑	§3; App. B
► **N3**	Cooperative Deadlock Prevention	m33	☑	§5
M5.3.2a	OS accelerator model	m24	☑	§4, 4.4.2; App. G
M5.3.2b	OS I/O bridge model	m30	☑	§4, 4.4.2
M5.3.2c	OS autonomous model	m39	☐	§4.4.2
M5.3.3	Investigate I/O support in microthreaded processors	m17	☑	§4; App. E & D
M5.3.4	Port I/O support to μTC	m22	☑	§4.2, 4.2.1
► **N4**	General OS Library Support	–	☑	App. G
M5.2.2	Integrate memory protection into processor emulator	m30	☐	[21] & §1.3

Key: ∞ → continual process, ► → new goal, ✳ → goal modified, ☑ → completed, ☐ → on-going

From the outset of the operating system work on the Apple-Core Project, the goals were necessarily fluid in nature. Table 1 summarises the work carried out, up to month thirty-three, in the domain of Work Package 5. The novel programming model and architecture being used have meant that the original milestones for this area of the project have, in many cases, been modified and/or redirected into new areas. This can be seen particularly with the addition of N1, N2 and N3 which were necessary objectives for the progression of the operating system infrastructure and in order to support work in other areas of the project.

Milestone M5.2.2, shown in grey in Table 1, is one other noteworthy modification to the operating system strategy. The previous Apple-Core deliverable ([21]) presented an in-depth description of the mechanics of memory protection in the Microgrid processor. Since this mechanism was devised and documented, it has been recognised that, within the scope of the project, to aid the progress of more fruitful areas, an emulated implementation of the memory protection scheme would not currently be necessary. This is largely due to the limited benefits that such emulation would bring; current, and indeed future, Microgrid applications do not require security of memory between applications. It is the intention of the researchers that memory protection *will* be implemented in the simulator, but at a stage when this is useful in terms of research, i.e. when the applications and operating system implementation are at a sufficient level of maturity to benefit from such a protection scheme.

2 Resource management

In a concurrent setting, resource management is both the partitioning of resources according to requirements, and the arbitration of access by concurrent processes. Besides the "usual" resources that need management, that is storage (disk and memory) and I/O channels, the following are relevant in a highly dynamic, concurrent environment:

- cores and clusters of cores,
- *execution contexts* for threads and families of threads, including the management of *thread local memory*.

We cover these aspects in the following subsections.

2.1 Spacial on-chip resource management

In the Apple-CORE architecture the execution resource visible to programs is the cluster, not the core. The cluster including all the co-processor and the on-chip interconnect is captured behind a single abstraction called the *SVP place*. Programs perform requests for allocation and deallocation of places based on computational requirements.

The overall intended interaction between programs and places is described in a previous publication [18].

The protocol proposed in this paper is defined hierarchically. At each level it defines an entity called a SEP, running on a specific site (i.e. core) at the local cluster. By placing the SEP at a specific site one ensures exclusion of access to its state (all requests go through the same core) and locality of state between different requests. Then programs make requests to the SEP for resources suitable to execute a named component, and the SEP "contracts" with the available resources the one that best matches the requests.

The intention of this framework is to recompute resource allocations in a fine-grained manner for each invocation of a resource kernel, possibly many times during a large computation that reuses the same kernel across multiple inputs in order to maximize the dynamicity of the system and minimize over-subscription of resources.

To achieve this in Apple-CORE we have implemented a lightweight SEP for clusters of cores on the Microgrid. We did not implement the "bidding" and "contracting" phases of the initial protocol specification since the homogeneity of the Microgrid allows to keep the knowledge about utilization in the SEP directly. Instead, the SEP thus controls the allocation state, by pooling identifiers for the available clusters at its level and performing best-fit allocations to satisfy requests from programs, which can specify their requests with either min, max or exact boundaries on the number of cores.

Details about the software interface are documented in CSA note [sl7] (attached as Appendix A). The SEP delivers the following information to programs:

- the actual number of cores in the allocated places,
- a place ID suitable to assign work by a *create* operation,
- architectural parameters like the number of family/thread entries per core.

During benchmarking, the following figures were extracted:

- Requests for place allocations are satisfied in a time logarithmic with the number of clusters managed at this level of SEP; less than 400 processor cycles per request with 8 different clusters (min 150 cycles, average 250);
- Requests for place deallocations are satisfied in constant time, less than 200 cycles per request (min 100 cycles, average 150).

These low latencies confirm that this implementation is suitable for a high-rate, fine grained space scheduling (millions of allocations cycles per second on a GHz-clocked chip).

2.2 Thread and family context management

Contrary to most software implementations, our architecture handles concurrency management in hardware. Both logical threads, families and their corresponding state are allocated, scheduled and freed using hardware processes on chip. As such, many aspects of a thread's execution context are allocated in dedicated memory structures in the cores instead of shared memory:

- the architectural register window is allocated per thread as a range of contiguous registers in a large register file (up to 32 registers per thread in a 1K register file);
- the instruction pointer, thread index, and other thread-specific constants are stored in a thread entry in a thread table (256 thread entries per core in the current configuration);
- the family termination status, index range, parent thread ID, initial program counter for all threads and and other family-specific constants are stored in a family entry in a family table (32 family entries per core in the current configuration).

The allocation of registers, as well as thread and family entries in the cores is entirely performed by the thread and family allocation/deallocation hardware processes, invoked as a results of the execution of the ISA extensions by the pipeline. They are all running asynchronously with the core's pipeline, and thus are long latency operations from the program's perspective.

These hardware aspects of concurrency resource management have been implemented mostly in the context of the previous project NWO Microgrids. However, in the context of Apple-CORE the following extensions have been designed and implemented:

- *thread local storage in shared memory.* Except in small data-parallel computation kernels or lightweight helper threads in complex algorithms, all of a thread's state rarely fits in the 32 architectural registers made available by the core's ISA. *Spilling* of intermediate results to memory is thus required in some fashion. To support this we have implemented automatic allocation of regions of the shared address space to each thread at an extremely low overhead, suitable for high-rate thread creation and termination. This is described below.
- *allocation feedback on resource exhaustion.* Since our system does not implement virtualization of concurrency contexts, there is a fixed upper bound on the concurrency resources that can be allocated by programs, possibly lower than the maximum concurrency expressed in algorithms. In other words programs may express the creation of more families threads than can be allocated on a given chip configuration. In order to prevent deadlock due to resource starvation, the thread allocation/creation process has been extended to report a status to programs when exhaustion occurs. This status can be used for deadlock prevention, as documented in section 5 of this report.

These two new features, combined with the hardware concurrency management previously available, form the low-level concurrency resource management of the Apple-CORE hardware operating system.

2.2.1 Thread local storage

In order to provide low-latency thread local storage, we have designed the following subsystem:

- the shared memory address space on chip is statically partitioned among all cores and thread contexts. With a 64-bit address space, this yields 65TB of usable address space per thread with security disabled, or 2GB of usable address space with security enabled;
- the cache protocol is modified to create cache lines on demand whenever a thread first accesses addresses in memory that corresponds to its own thread region in the address space. When a thread terminates, an invalidation message is broadcast for its own range of addresses that flushes all corresponding cache lines from the COMA network.

- only when the total thread-local storage requirement from all threads exceeds the capacity of the on-chip COMA network, are TLS cache lines spilled to external memory. At this point a dedicated MMU at the chip boundary is in charge of allocating storage in the external chip and mapping it virtually at the addresses required by the TLS cache lines.

Because the address space is statically partitioned and we assume power of two alignments of the various values involved in the partitioning, a base pointer to each thread's TLS can be readily computed from the core and thread ID by a small dedicated shift unit on the core's ALU. This way, this base pointer can be requested by each thread within one pipeline cycle.

In the compiler from our intermediate language μTC/SL, the TLS is then used as a usual stack frame for spilling registers and allocating "local" memory objects in each thread.

2.3 Memory storage

For the purpose of software global heap management, the external memory storage is managed using a two-level strategy:

- for small objects (under 256 or 512 bytes), an efficient, lock-less massively distributed concurrent heap allocation strategy is used;
- for larger objects, a hierarchy of shared global heap managers based on "traditional" algorithms is used.

The first allocation manager is new work contributed by Apple-CORE. It is currently detailed in CSA note [svp34], attached as Appendix F, and is the topic of a future publication. This protocol effectively allows lock-less, low-overhead completely concurrent heap allocation and deallocation from different threads. This small-object allocator is warranted due to the observation that most dynamically allocated objects (in number) are 256 bytes or smaller (2-4 words on average when compiling from functional languages); and that they commonly have very short lifetimes and are created/deleted highly dynamically and concurrently.

The second allocation manager used for larger objects is pool-based and can be configured in software. We currently use the public domain DLmalloc [20] implementation.

2.4 I/O Channels and other devices

This is expounded upon in section 4 of this report.

2.5 Summary

During this phase of this project we have focused on the key aspects of resource management on highly concurrent platforms: space scheduling over cores, and memory management. Through novel management schemes, we are able to provide low-overhead, low-latency fine-grained and scalable allocation of cores and memory to software components up to individual threads.

The aspects of I/O channel management are discussed later in this document.

3 Monitoring

This section describes support and findings about run-time monitoring in our on-chip highly concurrent architecture.

3.1 Introduction

We define monitoring as the action of observing the execution of programs without interference on their execution, for the purpose of extracting metrics and analyzing their behavior. More specifically in the context of Apple-CORE the following requirements have been recognized:

- *profiling of program sections.* This is the type of monitoring required e.g. to observe the performance of benchmark programs for the purpose of comparing different software implementations against each other.

  This requires us to observe different metrics in programs, from the point a program starts to the point a program terminates, and independently from other simultaneously running programs. Different parts in the execution of a program may need to be isolated with certainty, e.g. the initial data input, the final data output, and different iterations of a compute kernel.

  Also, when executing in accelerator mode or in the "slave" mode, care must be taken also to separate the work related to the initialization of the Microgrid from the effective program work.

  Because these various execution *phases* are defined by the structure of the software, there must exist some link between the input language(s), tool chain, and monitoring infrastructure.

  We call this type of monitoring *Synchronous in-program monitoring.*

- *profiling of overall architectural performance.* This is the type of monitoring required e.g. to observe the overall behavior of the hardware (utilization, load on interconnects, etc) under different software loads.

  This requires us to observe hardware metrics in a way that encompasses the current mapping of programs onto the architecture, i.e. including when different software components are sharing the same hardware, etc.

  We call this type of monitoring *Asynchronous architectural monitoring.*

To satisfy these two sets of requirements, we have designed principles and an infrastructure that is now integrated into our framework. They are described more technically in 3.2 and 3.3 below.

At this point we highlight that although these two sets of requirements appear orthogonal, they should be (and are already) used in conjunction. For example, when running a single program on the architecture, architectural monitoring produces an accurate picture of the effect of that specific program on the hardware. More importantly, by observing the overall load of one or more software component(s) on the specific sub-sets of the architecture onto which they are mapped, and correlating these results to the in-program measurements, one can deduce optimization parameters for future mapping decisions. We first suggested this idea in a recent publication [4] and then expounded the topic in a presentation [26] at the HPPC workshop at Euro-Par 2010.

In Apple-CORE, synchronous in-program monitoring is already used extensively by WP2 and integrated into Unibench. More information about this can be found in the WP2 report.

3.2 Synchronous in-program monitoring

For this type of monitoring we have designed and implemented a subsystem architecture as follows:

1. on every resource (core, cluster, cache), the concurrency run-time system allocates a region in the memory address space and ensures that these memory locations are asynchronously updated to reflect the current values of monitoring counters in the underlying resource. The

different counters include the cycle counter (for cores), individual (per core) and collective (cluster-wide) number of instructions executed, cache hits and misses, etc.;

2. the software stack defines and exposes a set of named pointers to these locations;

3. a software library is defined that contains services to:

   - copy the current values of a set of counters to another (temporary) location in memory; this is a *sampling* operation guaranteed to be side-effect free and to have an extremely low latency (a few cycles);
   - compute *differentials* between two samples that reflect the combined/average program behavior between the two sampling points;
   - *report* samples and differentials in a form amenable for further processing after the program has terminated.

We recognize a high similarity between this approach and the use of hardware performance counters in traditional architectures. There is sufficient similarity that the software API at point 3 could be made largely compatible with existing software using PAPI [6, 9] or PCL [5]. We chose however a simpler interface and to focus on the key aspects of the Apple-CORE project not usually considered by previous work: analysis of fine-grained concurrent behavior in programs.

This subsystem is intended to be used as follows: when designing a benchmark program, the program implementer is invited to insert calls to the software API to take samples at the start and end points of "interesting" sections in the program code. Then at some other point in the program code after the interesting sections are known to have finished executing, calls to report the values are inserted. By naming the sections with distinct (text) identifiers, the report can be then correlated back easily to the program sections.

By making each program responsible for its own monitoring (as opposed to e.g. report program events as an asynchronous execution trace and collect the trace afterward), we are able to encapsulate the monitoring state of a program instance using its own memory space. This is a simple abstraction that allows us to isolate easily the measurements taken by many instances of the same program running concurrently on different part of the architecture.

This design also allows for a feedback loop, where a program introspects its own performance and takes run-time decisions based on past observations. This theoretical possibility is intentional and reflects past knowledge in the field; it is not yet exploited in Apple-CORE but will be in the separate ADVANCE project.

3.2.1 Low-level support for performance counters

A key aspect of monitoring is to have as little impact as possible on the behavior of programs. Besides obvious considerations of performance, there is an additional issue with fine-grained concurrent systems: monitoring latencies of more than a few cycles could cause schedule discrepancies between runs with and without monitoring enabled, in turn causing side effects on synchronization, resource allocation, etc. With a system which requires monitoring functions to be called in software (syscalls or traps), the additional memory activity due to the call and moving arguments around would disturb cache and on-chip network behaviors.

We want to avoid these effects. In our chip architecture, we do so by implementing a lightweight form of memory-mapped I/O as follows:

- on each hardware device (core, cache, etc.) that can be monitored, we add hardware counters for each of the relevant metrics; these are updated by each basic hardware process involved. For example a clock cycle counter would be incremented on each clock edge; a cache miss counter would be incremented each time a request goes through a cache, etc.;

- we augment each core cluster's interconnect with a low-bandwidth, medium-latency ring network where each component with counters is connected. This network supports individual read requests to counters addressed by the node ID, counter ID, and collective accumulation of the same counter ID across all local nodes;

- on each core, we add a comparator to the pipeline memory stage, which transforms every memory load addressing a special address range into a network request to one of the counters, bypassing the normal cache access. This matches the high order bits of every address through a statically configurable pattern (we use all upper bits set to 0), and uses the low order bits as the counter address on the network.

Because we use cluster rings, the latency of counter requests to the local cluster are bounded by the size of the cluster, not the entire chip. Since our architectural design mandates clusters of 64 cores or less, we are able to bound the latency for these requests to 20 cycles or less.

Also, because these requests are handled like memory loads, they are asynchronous by nature and multiple requests can be issued before they are synchronized. This allows for efficient latency hiding in the sampling operations. Monitoring synchronicity occurs because these requests are not issued before known points in the program's execution, and they are synchronized before other known points are executed.

At the time of this writing, the following counters are supported:

- on each core:
  - clock cycle counter,
  - number of cycles the pipeline was active,
  - total number of instructions executed,
  - total number of floating-point operations issued to the FPU,
  - total number of completed memory load instructions,
  - total number of completed memory store instructions,
  - total number of bytes loaded (number of loads times the width of each load),
  - total number of bytes stored (number of stores times the width of each store),
  - integral of thread table occupancy (see below for definitions),
  - integral of family table occupancy,
  - integral of the allocation queue size for exclusive creates,
- at each L1/L2 cache:
  - number of cache hits,
  - number of cache misses,
- on each cluster:
  - number of cores on the cluster,
- at the external memory interface:
  - total number of cache lines fetched from external memory,
  - total number of cache lines stored to external memory.

As explained previously, all these counters can be sampled collectively across all nodes in a ring. For example, to get the total number of instructions, a network request can be sent across the ring to accumulate the number of instructions per core across all cores.

In this list the thread and family table occupancy merit special attention. The purpose here is to observe the utilization of the concurrency resources on chip, i.e. how much these tables were used between two execution points in a program. Because we cannot associate individual counters to each entry in these tables (e.g. number of cycles each entry was used), we needed to determine a metric that would allow us to compute utilization *a posteriori*. This is implemented as follows:

- when a table-updating event is handled by the relevant processes (family/thread allocation, creation, cleanup), a counter is updated which reflects how many family/thread entries are *currently* allocated;
- simultaneously, the product of the internal "current allocation" counter with the delay since last event (value of cycle counter at current event, minus copy of cycle counter taken at previous event) is added to the globally visible *occupancy counter*.

This process effectively *integrates* the allocation counter non-continuously over time. The average occupancy can then be obtained for an arbitrary interval of time by dividing this integral by the time elapsed, measured in cycle counts.

3.2.2 Software-hardware interface

The low-level hardware mechanism presented in the previous section is visible to software through the standard memory access interface, i.e. memory load operations. Each counter is identified by the low order bits in a memory address, i.e. we can abstract the set of all pointers through a set of named pointers to their pseudo-locations in the memory address space.

By further expanding the width of all counters to the width of an ISA register, exposing as a static constant (either in hardware or software) the total number of counters available, and ensuring that all counter addresses are continuous in the address space, we further homogenize the access from software by exposing all counters in a layout identical to a contiguous array in memory.

However the following information needs to be provided to software as well for each counter:

- a *label*, that will identify the counter in reports, etc;
- a *unit* type that provides safe guards against misinterpretations of the counter values; in particular we want to distinguish between watermarks, level values and accumulators/integrals.

For the time being, this information is expressed entirely in software, as a predefined (static) set of values specific to the Microgrid architecture. We envision this information to be available on chip through a configuration ROM.

See CSA note [sl8] for more details about how this interface should be used in programs.

3.3 Asynchronous architectural monitoring

The other requirement for monitoring is intended for situations where multiple programs are sharing resources which must be collectively monitored, or for the case where programs are not equipped with in-program monitoring, and thus must be externally observed.

For this we propose a straightforward mechanism: asynchronous, periodic sampling of monitoring counters. In this mechanism, a dynamically selected subset of all performance counter is collected across the entire network at a configurable interval, and streamed out of the chip onto a separate system in charge of collecting this trace and synthesizing high-level behavior metrics.

Because of the large latency of such an operation and the impact on the interconnect, we propose this sampling interval to be relatively large compared to the system clock period(s) (e.g. every 10 milliseconds on a GHz-clocked system). This contrasts with the previous approach which is more synchronous and allows for latencies of tens of cycles (10s of nanoseconds).

Also, because this sampling is asynchronous, the resulting measurements are accordingly *imprecise*: when collecting the instruction counts across the chip, the individual instruction counts on

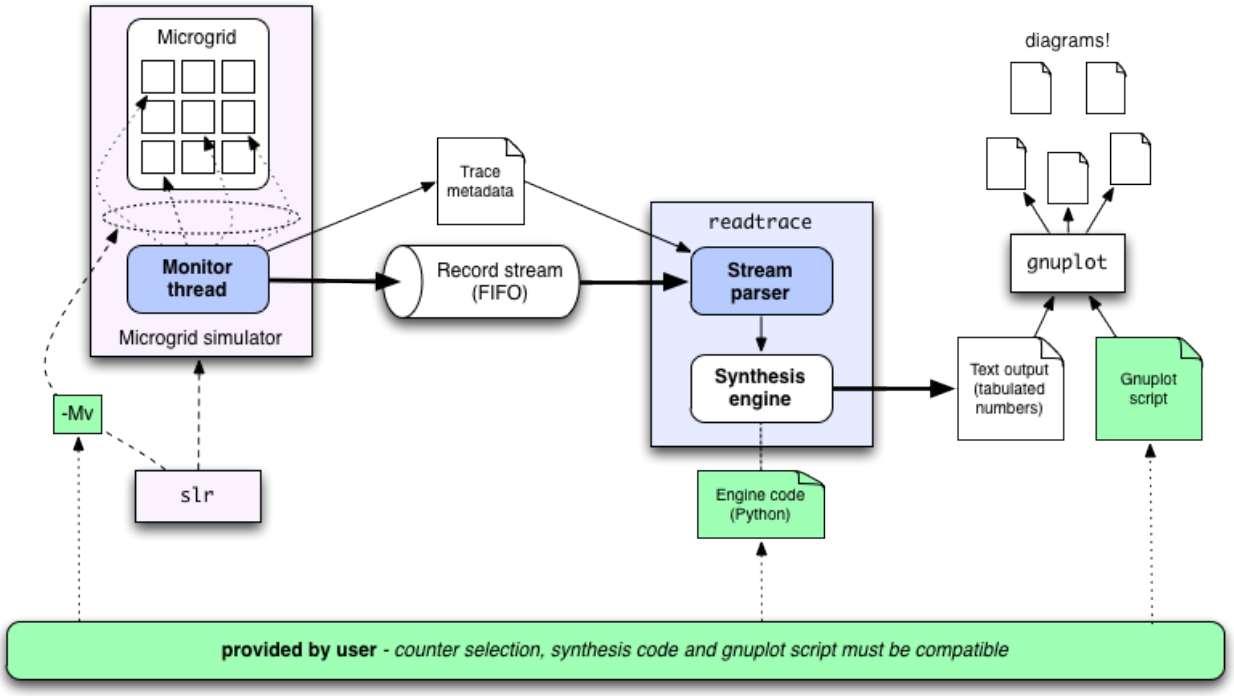


Figure 1: The monitoring architecture implemented by the Microgrid simulator and SL toolchain

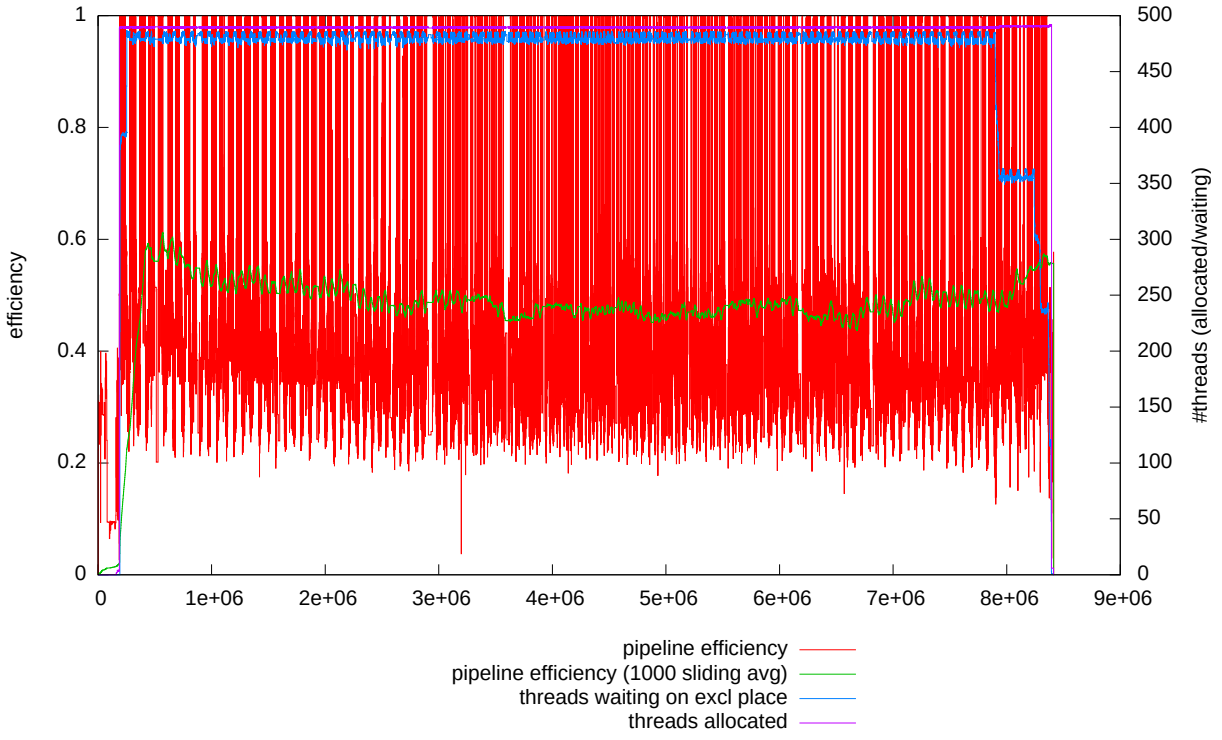


Figure 2: Example graph demonstrating the information made available by the architecture monitoring interface. The horizontal axis represents time in nanoseconds.

each cores will be sampled at different points in time, and thus the collective sets of counter values will not reflect an instantaneous observation of the chip. However, this imprecision is *bounded*, because the latency of this collection is bounded and the counters are evolving as a continuous function: one can estimate accurately the error margin of each sampled value by relating the local change rate of the counter with the overall sampling interval.

We have implemented this mechanism in our simulation environment as shown in Figure 1 with an example graph of the furnished data shown in Figure 2. More information about its function and its use is described in CSA note [mgsim9], attached as Appendix B.

3.4 Summary

During this phase of the project it became increasingly necessary to be able to monitor and introspect the management of concurrency on chip for both troubleshooting and analyzing program and architectural behavior. For this we designed two systems with low impact (next to none) on scheduling and performance, i.e. transparent to execution. One is controlled by programs to measure individual instances of kernels and the other to monitor asynchronously the architecture during execution.

4 Input/Output in SVP and Microgrids

4.1 Overview

This section presents the design for general purpose device input/output, and associated interrupt system, which works at both the level of the SVP concurrency model, in an *operating system* stack, and also with the associated hardware implementation in the many-core Microgrid. This section assumes a working knowledge of SVP and the Microgrid; appropriate background can be found in [16, 17]. The model and work presented in this section was published and presented at the SAMOS 2010 conference [13].

4.1.1 Motivation

As part of the AppleCore project, the Microgrid and SVP model are proposed not only as a solution for specialised scientific computation but also, importantly, as a system for general purpose computing.

The proposal of a more general purpose Microgrid has initiated great consideration and research into operating system features. Parallel architectures are necessitating a far more decentralised operating system design approach [31, 25], specifically for I/O, since it is not possible for every core to be directly coupled to the I/O infrastructure. As such it is desirable to have an I/O facility that is not merely an afterthought, but fundamentally engineered to make full use of the parallelism provided by a many-core system, the SVP/Microgrid model, for communication with a diversity of external devices.

4.1.2 Context of I/O Work

Figure 3 shows the areas of the AppleCore project that are affected by the work carried out on I/O. Bearing in the mind the I/O research included both software (OS stack) and hardware (emulated I/O cores), these areas namely include:

- An interface in μTC with which client functions can access the I/O subsystem (see subsubsection 4.2.1)
- An operating system stack which models I/O event and interrupts in the framework of SVP on the Microgrid (see subsection 4.2)
- An implementation in the Microgrid emulator of special 'I/O Cores' (see subsection 4.3)
- An architecture for modelling device interactions with I/O Cores, under emulation (see Section)

4.1.3 Related Work

The use of dedicated programmable processors to handle I/O is not something new, having been first introduced in 1957 when it was implemented in the IBM 709 system [14]. Following this development, the IBM System/360, System/370 and the architectures that superseded them have featured channel processors for high performance I/O [15]. Another system from this period that had even more similarities to the approach described in this report was the Control Data CDC6600 [29], which had a dedicated I/O processor that shared 10 distinct I/O processor contexts. As these I/O processors were very limited and only supported a simplified instruction set for handling I/O, they are not very different from the programmable DMA controllers found in modern computers. Both approaches serve the same purpose: to prevent the CPU from being frequently interrupted during dense I/O operations. This is also a problem in real-time embedded systems [27], where it is common that state-of-the-art micro-controllers are equipped with a peripheral control processor which can be used to handle interrupts while the main processor can still meet its real-time obligations.

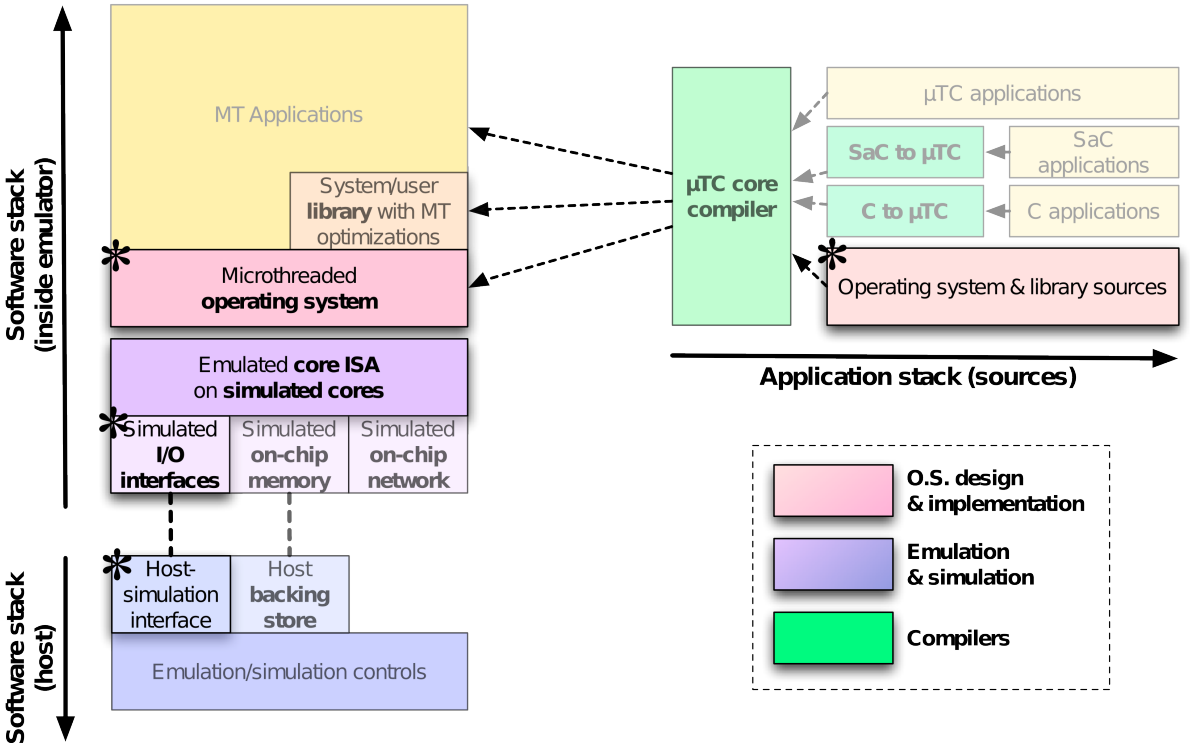


Figure 3: The location of I/O related investigation and implementation within the SVP model and AppleCore project. The areas discussed in this I/O section are highlighted and marked with an asterisk (∗)

The Helios OS [25] distributes small satellite kernels to programmable I/O devices in order to offload the execution of programs and system services. It uses affinity meta-data to hint about efficient placement of such programs to put data processing close to its source. As it targets heterogeneous platforms, it uses independent byte-code to represent programs which are then compiled for the specific device. In the I/O system described in this report, there is no such problem of heterogeneity as the I/O cores are similar to the other SVP cores, albeit with a restricted instruction set. The model described in this report would also benefit from the intelligent placement of components to use the locality of data. Others have also observed this as a problem, and suggest [7] that in order to achieve high throughput I/O, tightly coupled communication between the components, with no global interactions, is highly desirable. This suggestion mirrors the distributed operating system design approach being taken on the AppleCore project.

4.1.4 Key Areas of the Work

The designs and implementations described in this I/O section are a result of the novel methods with which software and hardware *Input/Output* communication and interrupts with external devices can be implemented within the SVP concurrency model and in the Microgrid hardware, but also more generally in many-core architectures and environments. The I/O scheme consists of novel implementations both at the abstract level of concurrency, making use of the features provided by SVP, in an operating system software stack, and also in the Microgrid many-core hardware, introducing the concept of an I/O core and the use of the efficient on-chip COMA memory system for high data-throughput with a variety of devices.

The implementation and modelling described is non-trivial and required great consideration and implementation time within the framework and feature set of SVP and the Microgrid, from both a software and architecture perspective.

The result is a highly scalable and parallel I/O architecture that, in the hardware implementation, bypasses external memory bus bandwidth limitations and contention, for efficient parallel

I/O. The model is completely decentralised from a monolithic operating system model, in common with the micro-kernel approach originally proposed for the AppleCore project. It embraces the idea of parallel services and as such is well placed for integration into a distributed operating system kernel.

4.2 I/O Operating System Stack

When implementing device I/O in the SVP model, and also on the Microgrid architecture, there are two distinct levels at which the problem is addressed. This section is concerned with the implementation of I/O at the software and SVP level, where the traditional requirements of device I/O must be reconciled with the advantages, facilities and restrictions of the SVP programming model. In practical terms, this level will reside in the operating system driver API.

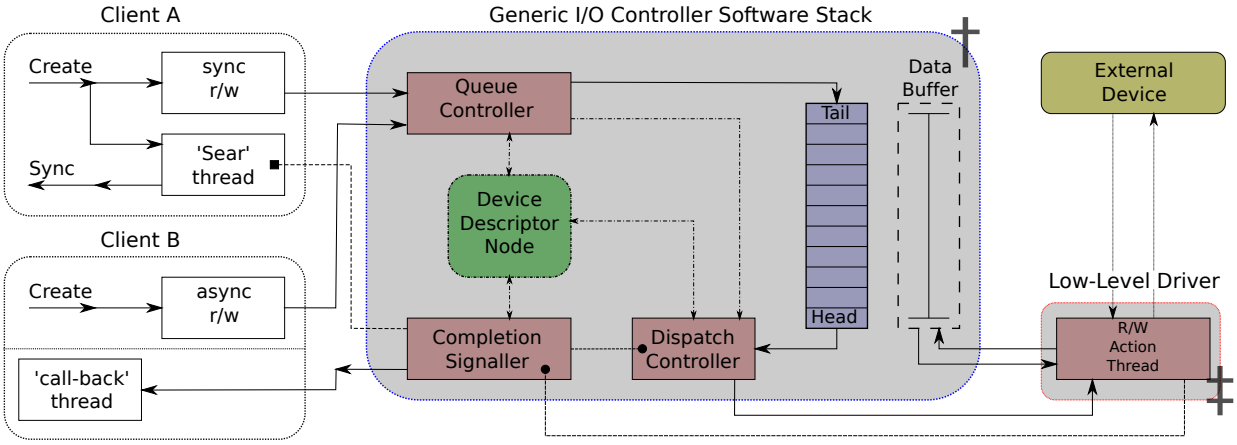


Figure 4: An overview of the Generic Software I/O Controller software stack which implements generalised I/O events using μTC for SVP. All processes inside the enclosure annotated by the † symbol must take place at the driver controller's designated *exclusive place*. Processes in the low level driver enclosure, annotated with the ‡ symbol, are executed at the designated SVP I/O place. Objects in square-cornered boxes represent threads; round-cornered boxes represent conceptual groupings and privilege domains (where Client A and Client B are applications).

The SVP software level I/O controller described in this section has been implemented in μTC for SVP; it is a highly flexible model of I/O which is compatible with all of the implementations of SVP, with extant interfaces to both the Microgrid hardware (see subsection 4.3) and also the μTC p-threads back-end (which was used for initial prototyping).

The I/O model implemented in SVP is designed to be as noninvasive as possible. This is achieved by providing a familiar software interface to client SVP programs: the standard 'read' and 'write' system/library calls are preserved and additionally the expected synchronous and asynchronous behaviour of I/O is provided. The example instance described in this chapter focuses largely on a device configuration of the 'request-response' form (e.g. a block device), since this is the most typical kind of device use-case, but the model is not limited to those types of devices.

It should also be noted that the model described in this section forms the generic basis of the I/O system. As a result, it is possible to encapsulate calls to this generic stack with further APIs, for instance a file system library which provides an abstraction over the structure of data on a device and would thus issue potentially multiple I/O actions for a single library call.

4.2.1 μTC I/O API

The API exposed to developers by the operating system stack is designed to be as familiar as possible, while still encapsulating the novel features and workings of the I/O architecture presented. With this in mind, the API consists of 'system calls' which initialise a device and allow for reading

```
//−−−−−−−−−−−−−−−−−−Configuration Methods−−−−−−−−−−−−−−−−−−−−−−−−
INIT_DEV(id, place, explace, flags, drv, buffer_size , q_size , nodepntr)

DEINIT_DEV(nodepntr)
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

READ_DEV(nodepntr, buffer, size)

WRITE_DEV(nodepntr, buffer, size)

READ_DEV_ASYNC(nodepntr, buffer, size, callback)

WRITE_DEV_ASYNC(nodepntr, buffer, size, callback)
```

Figure 5: Method signatures for the principle I/O API components (types omitted for simplicity), where: **id**→the channel ID of the device, **place**→the I/O place at which the device exists, **explace**→the exclusive place at which the higher-level device controller should execute,**flags**→some idiosyncratic device behaviour flags, **drv**→a pointer to the low-level driver for this device, **buffer_size**→the size of the internal device buffer, **q_size**→the length (in requests) of the internal queue, **nodepntr**→a pointer to the device descriptor node, **buffer**→the read/write data in memory for the I/O request, **size**→the requested read/write size of the I/O event, **callback**→the address of the callback method.

and writing to a particular device node (compare with reading and writing to file descriptors in Unix-like systems).

Figure 5 shows a summary of the main methods exposed to the system programmer by the operating system stack. 'INIT_DEV' is used to configure a 'device descriptor node' in memory, 'DEINIT_DEV' is used to release the resources held by a device, 'READ_DEV' and 'WRITE_DEV' are used to perform synchronous I/O operations, while 'READ_DEV_ASYNC' and 'WRITE_DEV_ASYNC' provide access to the asynchronous I/O operations. These methods are to be used in the standard C manner, as sequential method calls; they encapsulate the μTC thread creation and synchronisation behaviours in their expansion. An example program, using the I/O API shown above is included as Appendix G.

4.2.2 I/O Model (Synchronous and Asynchronous)

Figure 4 shows an overview of how synchronous and asynchronous read and write calls to the I/O subsystem work. This model is best explained by stepping through each component shown in Figure 4.

Client Request

Client threads A and B issue synchronous and asynchronous read/write requests, respectively, to the I/O API. By specifying the *Device Descriptor Node* for the device, this triggers entry into the generic driver I/O subsystem and a switch to system level privileges (similar to a system call) by the way of an SVP *create* action, which creates an instance of the appropriate library call thread at the *device controller place*. The desired action, a size and pointer to the data in memory are also provided.

Client A, issuing a synchronous read/write action, also creates an empty thread referred to as a *sear thread* which itself simply suspends indefinitely. Client A will then wait for this thread to complete by performing an SVP *sync* action. When the synchronous I/O action has eventually been serviced by the software I/O controller, the sear thread will be terminated by the controller issuing either a *remote shared-register write* (on the Microgrid) or an SVP *kill* action (on the PTL-backend). The concept of the sear thread is necessary to allow the synchronisation by a client over the I/O

event whilst still fully decoupling the execution of said client from the software I/O controller's context. This is due to a property of the SVP model, and other parallel models, which does not allow the synchronisation of two threads without a parent, child or sibling relationship.

Client B, issuing an asynchronous read/write action, provides the address of a *call-back thread* which will be created by the I/O software controller at the point of read/write service completion. Execution in Client B continues while the I/O operation takes place. It should be noted then that Client B should take care to account for the concurrent execution of its code by the asynchronous callback – the burden lies with the client to ensure synchronisation of its data-flow around this event.

Device Descriptor Node

The *device descriptor node*, created by initialising a device, is a data-structure in memory which contains the necessary information about the I/O device, pointers to the relevant data structures and the exclusive device controller place information. From the perspective of memory protection, this structure would reside in the system-level protection ring, accessible only to privileged code.

Request Queueing

The service executes the *Queue Controller* at the exclusive 'Device Controller' place. At this point, the semantics of the SVP exclusive place ensure that SVP creates are queued and that only one instance of the controller thread is executed at any given time. The queue controller now checks for space in I/O queue and, assuming there is a slot, adds the read/write request to the back of the queue. The queue controller then invokes the *Dispatch Controller* to notify it of a modification to the queue.

Request Dispatching

The *Dispatch Controller* examines the device descriptor node to see if an I/O operation is currently active. If an I/O operation is already active on the device, execution in the *software I/O Controller* terminates. Otherwise, it checks the pending queue of jobs. If the queue is not empty, it performs an SVP *create* on the *R/W Action Thread*, delegating it to the place at which the device exists and updates the device descriptor node to mark the device as active. The r/w action thread is created with the parameters of the particular read/write operation: the size, target, and a channel number. Execution in the *software I/O Controller* terminates and is fully decoupled from the low-level I/O action itself.

I/O Actions

In the *R/W Action Thread*, a single I/O operation from the queue is served and actual communication with the device takes place. The r/w action thread suspends based on the interrupt of the *External Device* while the operation is serviced by the device. The resulting data is read from or written to the appropriate memory location. Upon completion of the individual r/w, an SVP *create* action is performed on the *Completion Signaller*, to signal completion of the low-level operation. Execution in the *Low-Level Driver* ends.

Client Synchronisation Signal

The *Completion Signaller* thread, created by the completed r/w action, is responsible for 'waking up' the client. Based on the information in the record for the particular I/O operation, the completion signaller will either: terminate the appropriate *sear thread* in the client (allowing execution to continue in a synchronous I/O action) or perform an SVP *create* action on the specified 'call-back' thread in the client, for asynchronous I/O. The completion signaller also updates the Device Node Descriptor, signifying that the device is now 'free' and triggers the Dispatch Controller so that the next I/O operation can be processed.

General Notes

All device controllers in this model have to be initialised with a number of desired behavioural parameters. An important parameter is the location of the buffer for the I/O data. This can either be in the software device controller, where the appropriate I/O data will then be copied into the client's buffer as required, or the device controller can read/write the I/O data directly from/to the client's address space with each I/O action.

The previous example describes the *request-dispatch* device controller behaviour, where I/O requests trigger the dispatch of a low-level operation as required. However, the software I/O controller can also issue a continuously listening low-level thread which fills an internal buffer in the I/O controller stack. Subsequent I/O read requests can then be served from this buffer.

4.2.3 Low-Level Drivers

The low-level driver thread is required to perform a single I/O action directly with the device on the SVP place at which communication with the device can take place. Additionally, it is the low-level driver that generates the events which drive the higher I/O stack.

The low-level driver implements a standard interface which takes the necessary parameters for an individual device communication (for instance a read or write action, seek and also initialisation activities). Thus, only the low-level driver need be reimplemented to work with new device interfaces. The actual nature of the internals of the low-level driver depend on which SVP implementation is used. On the Microgrid, the low-level driver would need to perform register and bus level communication with the external device itself. However, were the distributed Pthreads SVP implementation in use, this low-level driver could simply be a proxy to the system calls of one of the host environments in which execution is taking place.

4.2.4 I/O Places

An I/O place (‡ in Figure 4) is an SVP place at which a particular device exists, in the sense that communication with that device can be locally achieved. An I/O place does not necessarily have to be remote, however in the Microgrid hardware implementation of SVP that I/O place will correlate to the I/O Core, introduced in subsection 4.3. In a more generic implementation of SVP, this I/O place could be a remote environment or machine that provides access to a particular resource. Importantly, more than one device may be associated with a particular I/O place – this is captured by the channel identifier.

4.2.5 Parallel I/O

The generic I/O model shown here, with the separation of atomic I/O actions from I/O places, makes parallel I/O relatively straight forward. If a particular device has many interfaces to the I/O infrastructure, where there are multiple places at which a low-level driver thread can communicate with a device, for instance in a RAID style configuration, then the *dispatch controller* can decompose a single I/O operation into multiple segments and distribute these to the array of I/O places. This approach is particularly useful when combined with lower-level implementation support, as is the case in the Microgrid (subsubsection 4.3.6).

4.3 Microgrid I/O Implementation

The SVP I/O subsystem described in subsection 4.2 requires support from the specific implementation; it must provide the *low-level driver* service through an interface to external hardware. This section describes how such support can be implemented in the hardware of the SVP Microgrid.

The overall hardware scheme for I/O in the Microgrid can be seen in Figure 6, where external devices are connected via a HyperTransport-like bus to dedicated *I/O Cores*. The scheme is based around a model of message signalled interrupts communicated via a bus interface, similar in nature to MSI in the PCI specification [28].

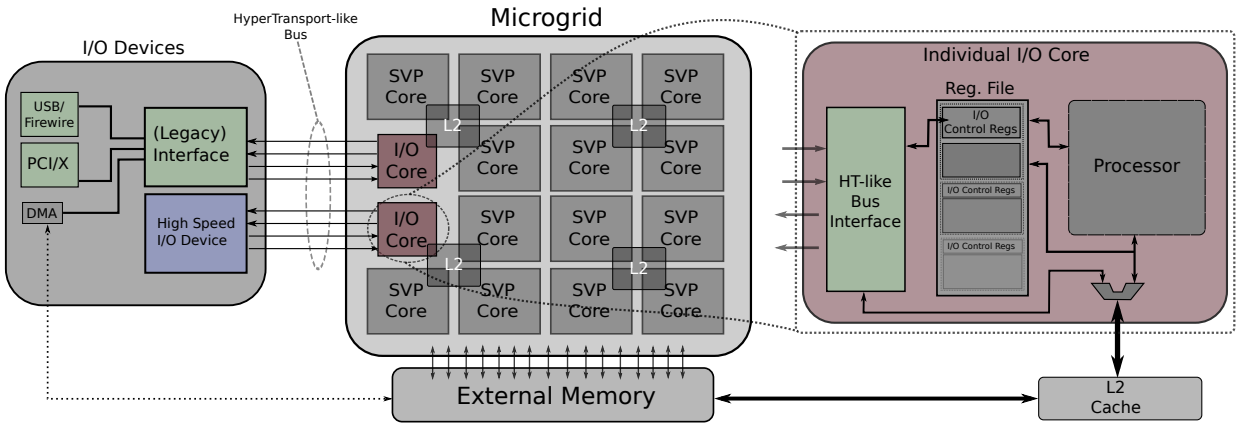


Figure 6: A schematic overview of the architectural implementation of I/O in the SVP Microgrid. An enlarged individual I/O core is shown to the right. The L2 Cache shown is part of the on-chip COMA. Individual external devices are shown to the left, including a potential legacy DMA controller.

4.3.1 I/O Cores

A derivative of the regular SVP core, the *I/O Core* is the hardware realisation of the conceptual *I/O Place* introduced in subsubsection 4.2.4. It is the place to which the *low-level driver* is created for a particular read/write operation. It can be distinguished from other general purpose cores in the Microgrid by the following characteristics:

Bus Interface – the I/O core contains a Bus Interface device which connects the I/O Core to a bus for high-speed communication with external devices. This hardware is only present in I/O cores.

Simplified logic – given the specialisation of the I/O Core as a place only for performing I/O, it does not need to contain floating point logic, and the size of the integer register file can also be much smaller than general purpose cores (given the simplicity of the threads it will be executing).

I/O Instruction – the pipeline of an I/O core responds to a special I/O instruction which allows threads to issue and wait for events on the high-speed bus.

The purpose of the distinct I/O core is to relieve fully-fledged Microgrid cores of the burden of I/O operations and to allow a higher density of parallel I/O to take place in the Microgrid. Specifically, the I/O core allows the rest of the Microgrid to continue executing in parallel while I/O operations are serviced. The reduced complexity of I/O cores, needing only to handle simple atomic I/O operations, means that their footprint in the architecture is relatively small, potentially allowing for more parallel I/O places in a fixed budget.

4.3.2 High-Speed Bus

Each I/O Core interfaces with a HyperTransport-like [8] packet/message bus by means of a simple on-chip bus controller. While in principle a variety of packet-based buses could be used, the HyperTransport bus was selected based on its ubiquity in various high-speed I/O applications and the ability to interface with a variety of existing high performance devices. As can be seen in Figure 6, the bus need not be connected directly to a single device, but can itself be connected to another interface which multiplexes between several devices using channel identifiers, including, for example, legacy bus implementations.

The bus is specified as HyperTransport-*like* because it is unlikely that all of the features of the HyperTransport specification would necessarily be implemented; rather, the hardware specifications of HyperTransport serve as a basis for achievable transfer rates in future simulation.

The width of the HyperTransport-like bus is variable and a parameter chosen at implementation time. The current HyperTransport specification (3.1) stipulates a maximum bus width of 32 bits

with a unidirectional transfer rate of *25.6 GB/s per second* (this can be seen as double to aggregate bidirectional transfers).

4.3.3 Bus Interface

The bus interface is a small piece of logic (comparable to a simple interrupt controller) present at I/O Cores which is responsible for connecting the processor pipeline with bus events and messages. The bus interface performs the following actions:

1. Compose and decompose bus messages appropriately, based on the associated channel information.
2. Deliver channel events into appropriate I/O control registers (subsubsection 4.3.5).
3. Perform reads and writes from/to memory of the data payload in I/O events at a specified location (this can be either to the memory subsystem directly, or to the register file).

The bus interface device is controlled entirely through the *ioctl* instruction. As noted in point 2, the bus interface is responsible for triggering changes in the state of the synchronising registers, based on bus events.

4.3.4 Device Communication and Interrupt Handling

Communication with external devices connected to an I/O Core take place using a special instruction, the execution of which is only defined at I/O Cores. The *I/O Control Instruction* (see Table 2) is used to perform an individual read or write operation on the bus and provides an associated synchronisation with this operation.

Table 2: I/O Core 'I/O Control Instruction'

Mnemonic	Operands (registers)		
ioctl	*control*	*size*	*src/dst*

Operands	Description
control	Specifies mode and channel (device) identifier
size	Requested size of read/write to perform to bus
src/dst	The target or source (buffer or register) of the read/write operation

The low-level driver described in subsubsection 4.2.3 essentially implements a small thread containing an appropriate (per device) usage of the I/O Control Instruction, associating the dispatched I/O operation's parameters to the operands of the I/O Control Instruction, after performing any required intermediate operations.

The *control* operand specifies a register containing bit-level information stipulating whether a read or write operation should be performed, the channel identifier (which identifies a device) on which to broadcast or listen and whether or not the bus interface should read/write data directly to the COMA memory interface or the register file (the latter negates the use of the *size operand*). This information is contained in one of the standard registers in an I/O Core.

The two modes of operation, Register$\Longleftrightarrow$Bus and COMA$\Longleftrightarrow$Bus, allow for the differences in performance trade-offs; Register$\Longleftrightarrow$Bus communication is useful for low-level control tasks where the control operation can be passed as an thread parameter and accessing memory is an unnecessary performance overhead. High-volume transfers can, in principle, be carried out through registers, however the implementation of the Bus Interface also permits a direct mapping of the processor's address space on reads and writes (see Figure 6 and subsubsection 4.3.3).

4.3.5 Synchronising and Interrupts

Synchronisation is achieved with an elegant modification to the semantics of Microgrid registers at I/O Cores. All registers in an core's register file in the Microgrid have a synchronising behaviour, achieved through the use of state bits at each register.

When the *ioctl* instruction is issued in the pipeline, the *control* register operand has its synchronising state marked as 'pending'. The low-level driver thread can then perform a read on this control register, at which point its execution will be suspended in the normal way of a thread which reads a register flagged as the target of a write operation. When the appropriate I/O channel's transfer completes, the I/O interface will adjust the state of the control register to 'full', at which point the suspended driver thread will resume execution. As shown in Figure 4, this would consequently trigger the *completion signaller*.

4.3.6 Memory Interface

All cores in a Microgrid are connected to the on-chip COMA hierarchy (see [17] for Microgrid COMA information). Data written to the Microgrid COMA memory system will migrate through the hierarchy to the location at which it is read. This property is exploited to achieve very high-performance I/O in the model described in this report.

All I/O operations, whether lightweight operations through registers or bulk transfers, will eventually write/read their data to/from the associated buffer in memory. This means that I/O is not bound by the traditional limitations of memory bandwidth, as is the case with existing DMA architectures [10], and is fully decoupled from external memory bus contention by being distributed to potentially several different I/O buses instead.

The organisation of the COMA memory system into rings, unifying separate caches, means that not only can I/O be extremely fast, but also extremely parallel. I/O operations can take place in parallel, at different I/O cores, with different clients, using only the local memory subsystem and avoiding thrashing of the memory hierarchy between conflicting operations. This property introduces the concept of *I/O locality*, where, for the highest performance, a place allocator will delegate a client to a place on the same COMA level as the associated I/O device's I/O Core; i.e. they will share the same L2 cache. A 'smart' placement algorithm would ensure that jobs on Microgrid are created at a place appropriate for the I/O dependencies of that particular job.

4.4 Summary

In this section we have presented the novel combination of methods for performing I/O in the highly parallel environment of SVP and Microgrids. The approach first explored the implementation at the level of the concurrency model; the way in which signalling and interrupts can be represented in the parallel environment of SVP, where conventional interrupt and I/O mechanisms are not applicable due to the decentralised nature of a many-core operating environment. The approach of using listener/writer threads with I/O places obviates the need for a central 'interrupt handler' model. This software level model has been fully implemented in μTC.

We also described the implementation of specialised 'I/O Cores' in the Microgrid hardware architecture. These cores are special processing units of reduced complexity which are connected to a high-speed bus for device communication, and they provide a device interface to the higher level I/O stack. The particular advantages of I/O cores are that they allow I/O operations to be fully decoupled from general purpose cores for truly parallel and scalable I/O. The ability of the I/O core to read and write data directly to the local on-chip COMA cache hierarchy allows for very high device-to-client transfer rates and bypasses the limitations of the standard DDR memory buses (which are already under intense pressure in a parallel architecture), when compared to conventional DMA.The hardware model of specialised I/O Cores is implemented in the Microgrid software emulator.

4.4.1 Performance Results

This section presents some simple performance results from experimentation with I/O cores in the Microgrid cycle-accurate emulator. These experiments were designed to highlight the scalability and parallelism of the I/O infrastructure described in the preceding model and not simply its raw transfer performance (which is simply calculable and uninteresting for experimental results).

Maximum Transfer Rates

The current specification [8] of the HyperTransport bus has a unidirectional bandwidth of up to 25.6 GB/s. If this were to be streamed into external memory, as with DMA, the bandwidth would be limited by the bandwidth of the memory. For state-of-the-art DDR3-1600 memory, this bandwidth lies at 12.8 GB/s. However, considering that external memory is shared by all processors on the Microgrid, the effective bandwidth is, in practice, considerably lower.

The COMA memory, with a 1.0 GHz cache-line-wide ring network (in a typical experimental Microgrid) can achieve a measured bandwidth of up to 64 GB/s. This bandwidth is guaranteed between local caches, without interference from the rest of the system. Thus, if the consumer of the data is physically close to the I/O interface, the HyperTransport bandwidth can easily be matched by the COMA system and, furthermore, these transfer rates can take place simultaneously at many rings.

Baseline Experimental Configuration

Table 3: Experimental Baseline Configuration

Parameter	Specification
Microgrid	Experimental Standard **128 Core** Micorgrid
Kernel Used	Modified version of **Livermore Kernel 11**
Problem Size	$\mathbf{2^{17}}$, double precision floating point
Subsequent Data Size	**2 MB** (two arrays used by kernel)
Kernel Execution Place Sizes	**8 places remote from I/O cores**, each of 2 cores (executing a parallel reduction), and **8 places adjacent to I/O cores**, each of 2 cores (i.e. sharing the same L2 cache of each I/O core)
Microgrid Frequency	**1.0 GHz**
L2-Cache Size	**32KB**
Memory Speed	**DDR 800MHz**
HT-Like Bus Frequency	**1.0 GHz**
HT-Like Bus-Width	**16 bits**
Number of I/O Cores	**1,2,4,8,16** (parameterised by experiment)
HT-line Buses at each core	**1** (currently a fixed property)
Simulated devices	Synthetic **SATA** devices, with average sustained transfer rate of **100 MB/s**

Table 3 shows the baseline configuration used for the results presented in the subsequent pages. The HT-like bus frequency and width were selected modestly to permit a realistic pin-out based on HyperTransport bus specifications at the highest numbers of I/O cores (where several buses are present). The simulated device, and associated bandwidth, were selected to show both scalability (before saturating the memory system) and also to show how the I/O subsystem can tolerate off-chip device latencies through I/O parallelism.

The kernel chosen is intentionally simple in nature, since the computation performance of complex programs is not to be demonstrated in these results. In the case of the selected kernel, its data is loaded through the I/O software stack (written in μTC) and the emulated I/O hardware (in the

cycle accurate emulator), using the specified number of I/O cores.

The following two sets of results firstly demonstrate the correct functioning of the I/O implementation at both the software and (emulated) hardware levels. Secondly, they demonstrate the two important performance properties of I/O on the Microgrid: scalability and COMA-cache-local I/O.

Scalability with COMA-Remote I/O

In this experiment, the baseline configuration shown in Table 3 is used, and the number of kernel processing places is held constant (to isolate first the effect of I/O scaling) at 8 places, each of two cores. The number of I/O cores is a variable of the experiment, and scaled from 1 I/O core up to 8 I/O cores, where I/O events are subsequently parallelised over the available I/O places (see subsubsection 4.2.5). Each executed experiment begins first by loading the required amount of data into memory, via the I/O cores, and then continues by processing this data with the kernel *elsewhere* on the Microgrid. The results are shown in Figure 7, where a dark and light bar represent a breakdown of the time spent in each run performing I/O and executing the kernel on the results loaded into memory.

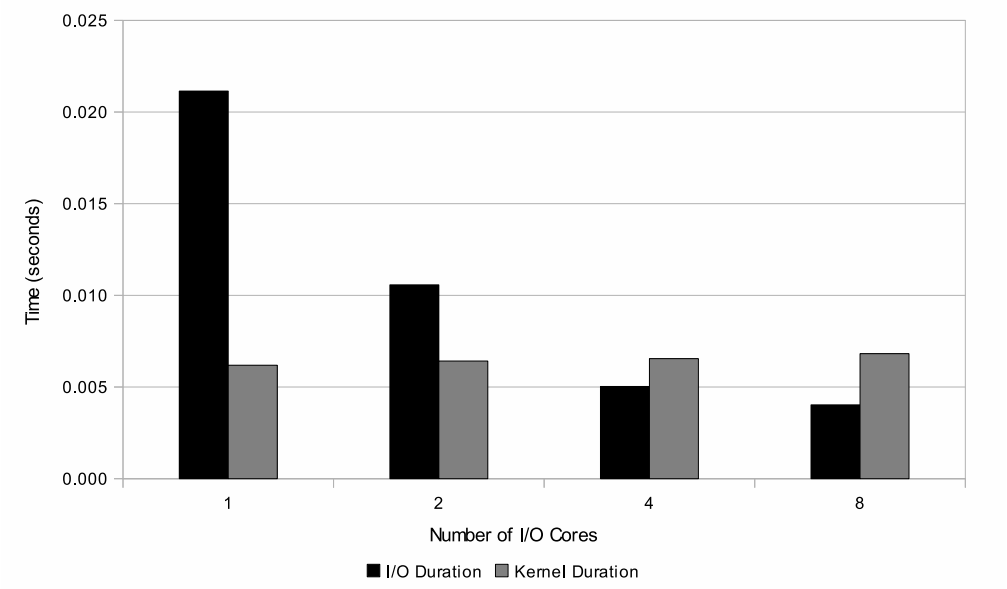


Figure 7: Emulated timing-results for I/O core scaling, with non-local kernel processing resources held constant

In the results presented in Figure 7 it can be observed that I/O duration initially dominates kernel execution duration, however the performance of the I/O subsystem scales well, almost linearly at smaller numbers of cores. As the number of I/O cores is increased, a slight slow down in scaling is observed as the overhead of I/O operations becomes noticeable. There is a slight increase in kernel execution time as the number of I/O cores is increased, likely accounted for by memory system behaviour as the number of caches used for I/O changes. The total amount of data read in these small experiments is still too large to be completely contained in the caches of the COMA memory system when I/O takes place in a single pass. As a result of this, the data will be written to the slower off-chip memory, and kernel execution duration begins to dominate I/O duration.

Scalability with COMA-Local I/O

In this experiment, the same baseline configuration is used as for the previous experiment but with one important difference: here, the kernel processing places are clustered, where each processing place is a pair of cores, with the I/O cores in order to share an L2 cache (there can be 4 clients for an L2 COMA cache, with the I/O core currently counting as two clients – one for the pipeline and

one for the bus controller). I/O transfers and kernel processing is arranged into blocks based on L2 cache size, where an I/O read fills the L2 cache, and then a kernel segment of appropriate problem (data) size is dispatched to the local processing resource as each I/O event completes. This process happens iteratively at each core and can be implemented elegantly using the asynchronous callback mechanism of the I/O model (see subsubsection 4.2.2). The results of this experiment are shown in Figure 8.

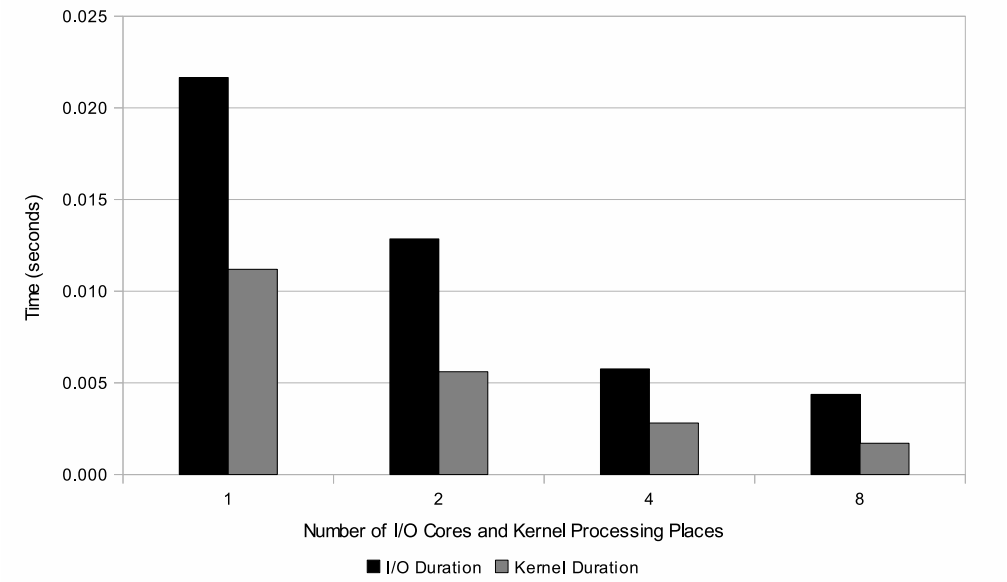


Figure 8: Emulated timing-results for I/O core scaling using blocked local cache transfers, and where cache-local kernel processing resources scale with the number of I/O cores

In the results in Figure 8 we can see roughly the same I/O scaling trend as in the previous experiment. However, total I/O time is slightly increased as there is a higher constant overhead, proportional to the number of times that the total transfer size is broken into cache-sized blocks (in the case of this experiment, each run issued 64 I/O operations; ≈2MB÷32KB, incurring the I/O stack overhead many more times than the previous experiment).

The interesting and important trend that can be seen in the local-I/O results is that kernel execution time scales very well; in the end, at 8 cores, the same kernel, executing on the same number of cores in the same organisation, gains a considerable speedup over the previous non-local version. This is because all of the I/O data is written to, and read from, an I/O-local cache – there is no need to process data from external memory or migrate cache lines to remote places on the Microgrid.

Results Conclusions

These encouraging results present only a small example of the problem space which is being examined during our investigation of I/O for SVP and the Microgrid. However, they serve to demonstrate the functioning and scalability of the I/O subsystem.

The I/O system of SVP and the Microgrid shown in this section is capable of absorbing the latency penalty of slower commodity devices, where they can be parallelised, and also to take advantage of much higher-speed devices by the use of COMA-local I/O, which can be shown to scale in the same way even when extremely high-speed devices are used at faster HyperTransport bus speeds. Indeed, such a software topology is necessary as COMA-remote I/O will be bound by internal and external memory bus and behaviour limitations.

A more thorough investigation of the I/O subsystem is currently underway, including analyses under various real-world application and load scenarios, with a view to further publication. Additional interesting points of discussion include the choice of the number of cores per L2 cache at I/O

places (potentially allowing for greater I/O processing locality, but greater cache contention) and also the size itself of L2 caches at I/O places.

4.4.2 Milestones

The work on I/O is mentioned specifically as a facilitator for two milestones. These milestones, along with the accelerator model, are discussed here, with particular reference to their relationship with this work.

M5.3.2a ✓ – OS I/O accelerator model: "The Microgrid can be used as a dedicated specialised processing unit by a host system." This Milestone was reached some time ago and this mechanism was used to produce the majority of emulation results for publication by the Apple-Core research teams. This is facilitated by the SL toolchain and the SL libraries (see Appendix G and technical note SL3).

M5.3.2b ✓ – OS I/O bridge model: "The Microgrid is coupled on-chip to a general purpose legacy computing core connected to legacy devices and running a legacy operating system and drivers," where communication between these entities takes place through an I/O channel. The IO research and implementation in this section fully facilitates this model and usage of the Microgrid by utilising an I/O device as a simple channel between host and Microgrid.

M5.3.2c ϟ – OS autonomous model: "The Microgrid is connected directly to I/O devices and runs hardware drivers natively." The model and emulated architecture implementation described in this section provides a standard interface, in C++, to which emulated devices (or device interfaces) can be 'connected'. Using this interface, any kind of device can communicate with applications running on the Microgrid, through I/O Cores and the operating system stack. Currently, a simple device exists which allows the Microgrid to access files on the host file-system; an emulated hard disk. All that remains for this milestone to be fully (and imminently) completed is the implementation of emulated devices and/or emulated device interfaces with an associated (simple) low-level driver object in the operating system stack (see subsubsection 4.2.3). Existing and recognised operating system libraries can be used to wrap the O/S stack and provide native support for things such as file-systems and networks. All of this work is a straightforward extension to the outcomes presented in this section.

4.4.3 Future Work

Current work is focussed on the quantitative analysis of the I/O model's performance in specific application scenarios using emulated devices, and the integration of the I/O stack into the mainline SL environment where it can then be used extensively in client applications.

5 Cooperative Deadlock-Prevention

5.1 Problem Description

In a microthreads program, every family create consumes resources; thread table entries, family table entries and registers. Since these resources are finite, situations can occur when a create tree becomes too deep and/or wide and there are no more resources available to perform a create. If no resources can be released without this create finishing, the create can never continue and resource deadlock has occured.

This problem is expanded by delegation. A microgrid is divided into clusters of cores, each with their resources. A create must be targeted to a specific cluster. If there are no more resources on the create's target cluster, even though resources may be available at other clusters, the create cannot continue and resource deadlock can still occur.

There are some solutions to solve these issue in certain cases. One is to perform a static analysis of the concurrency tree and request enough resources for the worst case scenario. This solution, when it can be applied, is quite powerful as it completely alleviates the need for the deadlock prevention mechanism described below, reducing the size of the code.

The delegation problem can be solved by not sharing places between applications with no knowledge of each other. Then, if the requester of a place knows exactly how many resources this place contains, it might know exactly if and when those resources will run out. This allows the compiler or runtime to perform optimizations such as adjusting the block size to prevent this deadlock from happening.

However, given that the number of resources available to an independent program can vary at run-time and given that we want binary compatibility of programs across a range of configured microgrids, programs would ideally have to be written to acquire knowledge of the available resources and their division into clusters at runtime and then issue the create accordingly. Unfortunately, for modular programs or programs generated from higher-level languages, such knowledge is non-trivial to obtain or act upon.

To combat the threat of resource deadlock in these cases where knowledge about usage of resources is limited or non-existance, a general method is desired that will ensure deadlock freedom in all cases, with a possible sacrifice of performance.

5.2 Sequentializing

The immediately available solution to avoiding resource deadlock for creating families of threads is not to do it. If, when resources run out, a call to the sequential version of the same code is substituted for a create, this code can continue to run in the single thread context of the parent, using a traditional stack, which can typically support a much deeper call tree than on-chip resources.

So, if we assume cooperative deadlock prevention among all code, we can define the following code transformation whenever a create needs to be done, given that we have the pointers for the sequential and threaded version of the target code. The transformation turns the following code:

```
create(F, Place, Start, Limit, Step, Block) work(...);
B;
sync(F);
```

into:

```
F = allocate(Place);
if (F == 0) {
   B;
   for (Start, Limit, Step) {
      work_seq(...);
   }
} else {
```

```
    do_create(F, Start, Limit, Step, Block) work_thread(...);
    B;
    sync(F);
}
```

The allocate operation will return 0 if the destination place is out of resources, and the creating thread will then execute the sequential version of the algorithm instead. Note that the sequential version is effectively run at the point of sync, i.e., the body between create and sync is run first. This has to be done because the body B can write shared and globals that are required by the family. This transformation also applies, recursively, to B.

5.3 Registers

The downside to this approach is that it generally requires more registers. When threads are compiled with an optimized register context suited to their exact requirements, a function call cannot be made because the callee shares the register context of the caller and the compiler cannot compile the callee for an unknown register context layout. Thus, every thread that has a function call must be compiled with a fixed register layout to allow for function calls to be made.

Note that this only applies to threads that have function calls due to this deadlock avoidance protocol. Threads without creates, or 'pure' creates that are not subject to this protocol, or those with inlined function calls, can still have optimized register contexts.

5.4 Group creates

For the above construct to work, the allocate operation must atomically allocate enough resources on every core that the family will run on. For local create this is easy, as it only runs on the core where the allocate is executed. For delegated creates, a network message will effectively perform a local create at the destination core. The issue arises with group creates which need an entry on every core. While a message could be sent around with an undo message if the allocate fails, this will cost many cycles. A more efficient solution is to use a single place-wide combined signal that indicates the availability of at least one free context on every core in the place, and a place-wide signal that is used to reserve one context at every core in the place.

These combined single-bit signals, which each must be able to traverse the entire cluster within a cycle, allows a core to determine whether or not a group allocate can succeed.

Note that a second global signal is also required to indicate that a group create has been made, in order to have each core reserve one context and adjust their 'free contexts' signal accordingly for the next core that wants to do a create. This avoids situations where another core wants to do a group create without the former group create having reached all cores, thus causing a false 'free contexts' signal to be interpreted.

5.5 Delegated creates

As mentioned above, delegated creates simply use a traditional try-fail approach. If the remote create cannot acquire a context on the remote core, the family is run sequentially on the parent core.

5.6 Exclusive creates

Unlike group and normal delegated creates, which can default to run sequentially on the parent core if the creates 'fail', this cannot be done for exclusive creates, since their semantics depend on being run at a specific place.

Fortunately, there is an easy solution: on every core, reserve a single context for exclusive creates. Since only one exclusive create can be active on a core at any given time anyway, this solution ensures that exclusive creates can always be run, as long as no cycles exist in the target places of exclusive creates (which can be guaranteed in software).

Note that the check for free contexts in the mechanisms outlined above must not consider this reserved entry since they cannot use it.

5.7 Hardware extensions

To enabled cooperative deadlock prevention, two things are required from the hardware environment. First, the allocate operation must fail if it cannot guarantee that the family has enough resources to be executed. Failure can be indicated by returning a certain "illegal" value (e.g., 0).

Secondly, guaranteeing availability of enough resources must mean, specifically, that the hardware guarantees that on every core where the family will (or can) run, there is:

- At least one free family table entry
- At least one free thread table entry
- At least N free registers for every type, where N is the maximum number a thread can use, e.g., on the Alpha, this means 32 free integer registers and 32 free floating-point registers.

5.8 Summary

In this section we outlined a protocol for running code with unknown resource requirements, on a platform with possibly unknown resources availability. This was a critical requirement uncovered in the previous phase of the project. The protocol involves using a function call as a fallback strategy should not enough resources be available for a create. To this end, the create operation must fail gracefully when resources cannot be allocated, and every thread that makes a function call must be compiled with the same register layout. This solution is a last-effort fallback solution, as other optimizations should be tried first, such as static analysis of the code combined with reserving entire places to allow the compiler or run-time to guarantee that deadlock will not occur.

6 Report Summary

This report has described in detail the main areas of progress within Work Package 5, the porting of operating system facilities to SVP and the Microgrid. The progress documented in this report accounts for an extensive amount of work, but can be briefly summarised into the following key areas:

Resource Allocation – the implementation of a fully-fledged resource allocation system for both processing and memory resources.

Run-time Monitoring – extensive state and performance information available dynamically to programs undergoing execution.

Input/Output – the investigation and implementation of both a software and hardware level system to facilitate scalable I/O in SVP and the Microgrid

Deadlock Prevention – a hardware/software codesigned system to bypass the serious hurdle of resource deadlock in SVP/Microgrid applications

Library Support – extensive implementation of standard library support for client applications

Together, the above work constitutes the solid foundation of the operating system features required by the Microgrid as a processor for general purpose computation.

The subsequent appendices are included to support the work in this report and should be used for additional information, particularly where directly specified in the body of this document.

References

[1] Design principles for end-to-end multicore schedulers. In *2nd Workshop on Hot Topics in Parallelism*.

[2] Virtual organization support within a grid-wide operating system. *IEEE Internet Computing*, 12(2), 2008.

[3] Apple-CORE Description of Work, Annex I (Version 1.6), September 2009.

[4] Thomas Bernard, Clemens Grelck, Michael Hicks, Chris Jesshope, and Raphael Poss. Resource-agnostic programming for many-core microgrids. In *Proc. 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, Ischia - Naples, Italy, September 2010.

[5] R. Berrendorf, H. Ziegler, and B. Mohr. Pcl-the performance counter library: A common interface to access hardware performance counters on microprocessors. *Central Institute for Applied Mathematics, Research Centre Jülich GmbH*.

[6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. 2000.

[7] Matthew Burnside and Angelos D. Keromytis. High-speed I/O: the operating system as a signalling mechanism. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 220–227, New York, NY, USA, 2003. ACM.

[8] HyperTransport Technology Consortium. HyperTransport I/O Link Specification. Technical Document HTC20051222-0046-0026, July 2008.

[9] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems.

[10] A. F. Harvey. DMA Fundamentals on Various PC Platforms. Application Note 011, National Instruments, April 1991.

[11] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The mungi single-address-space operating system. *Software?Practice and Experience*, 28(9):901 – 928, 1998.

[12] Michael A. Hicks, Chris Jesshope, Mike Lankamp, Raphael Poss, and Michiel van Tol. Towards a microgrid hardware i/o mechanism. SVP Note SVP31, University of Amsterdam, 2009.

[13] Michael A. Hicks, Michiel W. van Tol, and Chris R. Jesshope. Towards Scalable I/O on a Many-core Architecture. In *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, pages 341–348. IEEE, July 2010.

[14] IBM. 709 data processing system. `http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html`.

[15] IBM. Ibm system/370 principles of operation, 1974.

[16] Chris Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008.

[17] Chris Jesshope, Mike Lankamp, and Li Zhang. The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News*, 37(2):38–45, 2009.

[18] Chris Jesshope, Jean-Marc Philippe, and Michiel van Tol. An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the SVP model of concurrency. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 218–228, 2008.

[19] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *ASPLOS-V*, pages 175 – 186, 1992.

[20] D. Lea and W. Gloger. A memory allocator, 2000.

[21] J. Masters, M. Lankamp, C. Jesshope, R. Poss, and E. Hielscher. Report on memory protection in microthreaded processors. Deliverable D5.2, Apple-Core Project, December 2008.

[22] Joe Masters, Mike Lankamp, and Raphael Poss. Generalized i/o events for the microgrid. SVP Note SVP21, University of Amsterdam, 2009.

[23] Joe Masters, Raphael Poss, Michiel W. van Tol, and Chris Jesshope. Generalized synchronization in svp. SVP Note SVP20, University of Amsterdam, 2009.

[24] Andrei Matei. Towards adaptable parallel software – the Hydra runtime for SVP programs. Master's thesis, Free University of Amsterdam, 2010.

[25] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[26] Raphael Poss. Resource-agnostic programming of microgrids. `http://www.hppc-workshop.org/HPPC10-Poss.pdf`, September 2010.

[27] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 167–174, New York, NY, USA, 2009. ACM.

[28] PCI SIG. PCI Local Bus Specification Revision 2.3 MSI-X ECN. `http://www.pcisig.com/specifications/conventional/msi-x_ecn.pdf`.

[29] James E. Thornton. The cdc 6600 project. *Annals of the History of Computing, IEEE*, 2(4):338 –348, oct.-dec. 1980.

[30] M. van Tol. Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, 2006.

[31] David Wentzlaff, Charles III Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. A Unified Operating System for Clouds and Manycore: fos. Technical Report MIT-CSAIL-TR-2009-059, Computer Science and Artificial Intelligence Lab, MIT, November 2009.

[32] Mett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS-X*, 2002.

A SL Library: dynamic place allocation (TR)

SL Library: dynamic place allocation

Key: sl7
Date: 2010-06-17
Status: Draft
Author: Joe Masters
Author: Raphael 'kena' Poss
Author: Chris Jesshope
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/sl7.txt
Version: sl7.txt 3860 2010-06-17 21:44:02Z kena

Abstract This note documents the work on place allocation initially prototyped by Joe Masters and later integrated as proof-of-concept in the SL toolchain.

Introduction / vocabulary

We identify the distinction between:

1. SVP *place identifiers*, the opaque 64 bits of data that are required and sufficient for the delegation of SVP's *create* action; and

2. place "identifiers" as opaque handles to a group of resources that is allocated from a common pool, and whose allocation needs to be tracked by e.g. an operating system.

Whereas the former entity contains only the information required to perform a *create* operation, the latter needs to embed the following extra information for use by programs and libraries:

- amount of parallelism (cores, family entries, thread entries, etc);

- whether the place is exclusive;

- optionally, whether the place is shared between multiple components, and other possible future attributes.

This list of "place attributes" may be extended in the future, for example to embed topology information such as the "cost distance" to other places.

For the sake of clarity, we will keep the name "place identifier" for the low-level entity used to perform the *create* action; and introduce the term *descriptor* for the high-level entity which carries the additional, software-level information. This information is expected to be stored in shared memory, and a memory pointer to such a descriptor can be termed a *handle* to that descriptor.

Next to these concepts, we introduce the concept of *SEP* from the AETHER project. A SEP is an abstraction of a "place manager", i.e. a combination of:

- a dedicated set of resources,

- a coordinated internal state,

- services which provide allocation from the dedicated set to client components/threads.

By design SEPs can be stacked hierarchically by allocating a set of places from one level and using it as a dedicated pool for the levels underneath. This is not discussed here, see Future directions below.

Overview

We propose an implementation where the 64-bit SVP *place identifier* is an attribute of a place descriptor. This implementation manages and allocates descriptors for programs, which then can extract place identifiers from them for use in *create* actions.

> **Note**
>
> This can be opposed to e.g. an implementation which tracks descriptors only internally, and instead hands out place identifiers to programs. In such an implementation it would be possible to map back a place identifier to the descriptor for the purpose of answering queries about the number of cores, owner, etc. (e.g. using a hash table). We choose to avoid this type of implementation since the cost of such queries on the Microgrid seems higher than the two memory loads required by the descriptor approach.

Management of descriptors is encapsulated to allow hierarchical definition of SEPs. We propose the following 3 services as a starting point:

- *place allocation*, where programs request a place to a SEP and specify a policy for the allocation;
- *place deallocation*;
- as a mean to troubleshoot / illustrate the mechanisms and their uses in programs, a way to print out the current allocations for a given SEP.

These services can be indicated by a combination of a place identifier (where the services should be created as a thread family of one thread), a pointer to a thread function (the instructions to perform the task) and an opaque data structure in memory (for internal use by the SEP), all of which can be encapsulated behind a single address in memory.

Application programming interface

Programs making use of this implementation should include the standard SL header `<svp/sep.h>`.

After including this header the preprocessor macro `SVP_HAS_SEP` is set to a nonzero value if the API described below is available.

> **Note**
>
> Programs should be written in such a way that the special place identifier `PLACE_DEFAULT` is used for creates whenever `SVP_HAS_SEP` indicates that place allocation is not available.

Data types

SVP place identifiers have the pervasive, opaque data type `sl_place_t` in SL. This is unchanged.

Place descriptors however are encapsulated in the type `struct placeinfo`, with at least the following fields:

```
struct placeinfo {
   sl_place_t  pid;      /* place identifier for creates */
   sl_place_t  soft_pid; /* "soft" place identifier for creates */
   uint16_t    ncores;   /* number of cores */
   uint16_t    nfamilies_per_core; /* FTEs / core */
   uint16_t    nthreads_per_core;  /* TTEs / core */
   bool        shared;   /* whether the place is used by multiple clients */
   bool        exclusive; /* whether the place is suitable for exclusive creates */
};
```

Note

There may be additional data in memory appended for internal use by the SEP, and the length of this additional data is not known to programs. Therefore programs cannot copy this structure in memory and should instead copy pointers to it.

Note

The field `pid` contains a place identifier such that a create to that place always *effectively* delegates the work to the target place, possibly waiting until the place becomes available. In contrast, a delegate to `soft_pid` may fail if the target place is busy, and allow the program to inline the computation instead.

SEP encapsulation

SEPs are encapsulated in the type `struct SEP`, defined with at least the following fields (TC syntax):

```
struct SEP {
   /* place identifier where to place SEP calls */
   sl_place_t  sep_place;

   /* thread function for allocation */
   thread void (*sep_alloc)(struct SEP*,
                            long policy,
                            shared struct placeinfo* result);

   /* thread function for deallocation */
   thread void (*sep_free)(struct SEP*,
                           struct placeinfo*);

   /* thread function for status */
   thread void (*sep_dump_info)(struct SEP*);
};
```

This defines a data structure in memory whose first field is a place identifier, and the subsequent fields pointers to thread functions for respectively allocation, deallocation, and (optionally) console output of troubleshooting information.

Note

As for `struct placeinfo`, here may be additional data in memory appended for internal use by the SEP, and the length of this additional data is not known to programs. Therefore programs cannot copy this structure in memory and should instead copy pointers to it.

Root SEP

When the environment is set up, a pointer to the root SEP is accessibly by programs and declared in `<svp/sep.h>` as follows:

```
extern struct SEP *root_sep;
```

Allocation policies

Allocation policies are specified using the OR combination of the following specifier:

- specifiers about the number of cores (ORed together with the desired number of cores):

  - SAL_MIN: the provided place must have at least the number of cores specified;
  - SAL_MAX: the provided place must have at most the number of cores specified;
  - SAL_EXACT: the provided place must have exactly the number of cores specified;
  - SAL_DONTCARE: any place size will do.

- exclusion: SAL_EXCLUSIVE;
- agreement to sharing with other requesters: SAL_SHARED.

> **Note**
>
> it is expected that applications will tolerate sharing for exclusive places, where work is expected to terminate within a finite amount of time; this contrasts with non-exclusive places where there may be performance/availability requirements, in which case sharing is undesirable.

Performing SEP calls

The actual thread function to perform services is dependent on the SEP being used: the function pointer is stored in the SEP data structure. However, since such SEP "method implementations" can be reused across SEPs, they also take as *argument* a pointer to the SEP being invoked.[1]

Also, as the services coordinate around some internal state, all "calls" (creates) to the API must be performed using the same place identifier, using exclusion. This place identifier is available as the first field of the SEP abstraction.

Finally, all services should be invoked by creating a family of a single thread.

Therefore, all calls to SEP services look like the following (TC syntax):

```
create(;sep->sep_place;;;;;) /* exclusive */
   sep->SERVICE(= sep, ARGS...);
sync();
```

or, using SL:

```
sl_create(,sep->sep_place,,,,, sl__exclusive,
          *sep->SERVICE,
          sl_glparm(struct SEP*, , sep),
          ARGS...);

sl_sync();
```

Place allocation

The allocation method sep_alloc takes a policy as input, and returns a handle through its shared argument.

The shared argument can be initialized to any value.

If the allocation fails, the returned handle is set to 0.

Example use (TC syntax):

```
create(;sep->sep_place;;;;;) /* exclusive */
    sep->sep_alloc(= sep,  SAL_MIN | 4, p = 0);
sync();
if (p == 0)
   die("place allocation failed!");
```

[1] this is much like the this pointer is passed implicitely as an argument to object methods in C++.

In this example, the client program creates a family of one thread running exclusively at the place indicated by the `sep_place` field in the data structure pointed to by `sep`; provides the pointer `sep` also as first global argument; indicates the `SAL_MIN` policy with a requested number of 4 cores; and defines a shared thread argument `p` to hold the returned handle, initialized to 0. After synchronization, the handle is tested to check that the allocation succeeded.

This code is equivalent to the following SL syntax:

```
sl_create(,sep->sep_place,,,,, sl__exclusive,
          *sep->sep_alloc,
          sl_glarg(struct SEP*, , sep),
          sl_glarg(unsigned long, , SAL_MIN | 4),
          sl_sharg(struct placeinfo*, p, 0));
sl_sync();
if (sl_geta(p) == 0)
   die("place allocation failed!");
```

Place de-allocation

The method `sep_free` deallocates a place identified by a handle.

Example use:

```
create(;sep->sep_place;;;;;) /* exclusive */
   sep->sep_free(= sep, = p);
sync();
```

Same code in SL:

```
sl_create(,sep->sep_place,,,,,sl__exclusive,
          *sep->sep_free,
          sl_glarg(struct SEP*, , sep),
          sl_glarg(struct placeinfo*, , p));
sl_sync();
```

Status information

The method `sep_dump_info` prints information about the places managed by the SEP on the standard console output. It does not change the state of the SEP.

Implementation notes

The prototype (proof of concept) implementation provides a root SEP which buckets the hardware places by size; and then allocates by searching the buckets nearest to the size specified in the order specified by the policy.

No assumption is made that places have sizes that are powers of two, so the allocation algorithm must often scan through empty buckets up to (or down to) the next available size when an exact match is not found.

Future directions

The current implementation does not provide the actual service for SEP derivation (making a new SEP out of a subset of allocated places).

It seems also interesting to identify the "owner" of places during allocation for debugging purposes, and later for security purposes.

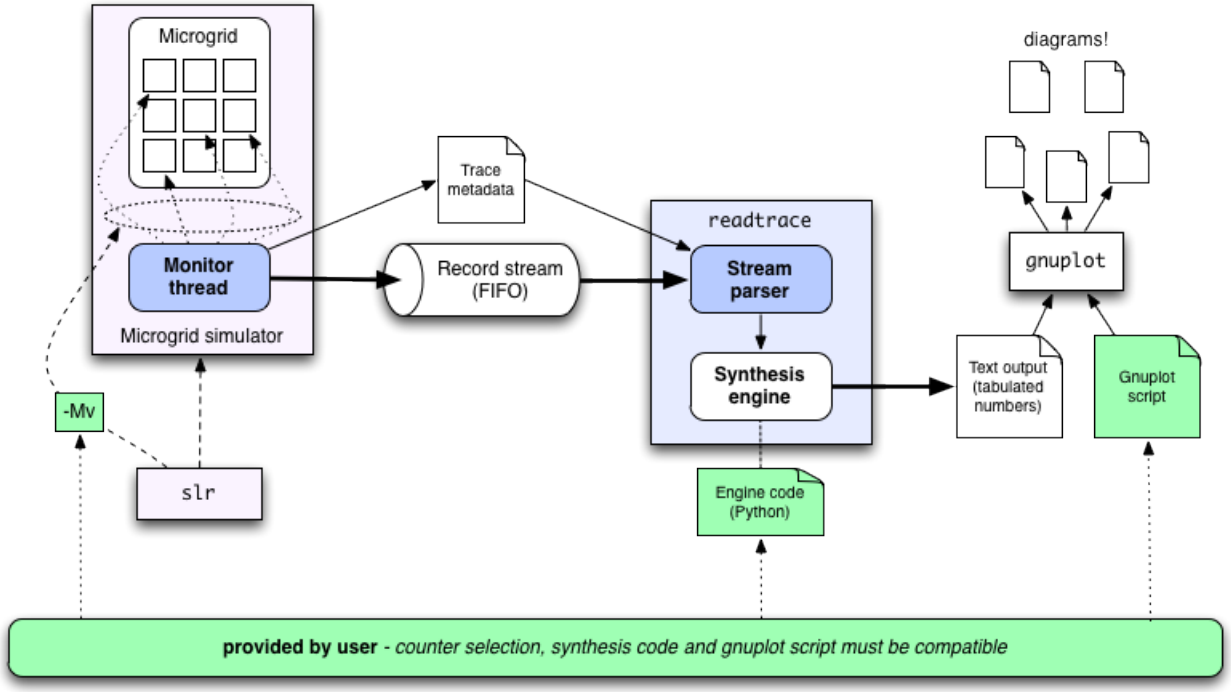
B Asynchronous simulation monitoring (TR)

Asynchronous simulation monitoring

Key: mgsim9
Author: Raphael 'kena' Poss
Date: 2010-09-02
Status: Draft
Version: mgsim9.txt 4027 2010-09-02 12:48:12Z kena
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/mgsim9.txt

Abstract This note describes the simulation monitoring framework introduced in summer 2010.

Overview



The diagram above summarizes the framework:

- the Microgrid simulator is augmented with a *monitor thread* which reads the simulation state asynchronously and redirects the information outside of the simulation in a binary format;
- a new tool `readtrace` is introduced, which connects a *stream parser* with a user-provided *synthesis engine*. The stream parser reads the binary records produced by the simulation monitor and passes them to the synthesis engine to be transformed.

The synthesis engine should then produce textual, tabulated decimal numbers suitable for input by a diagramming tool such as GNUplot.

Extra features

The following features have guided the design:

- the samples can be piped through a FIFO buffer; this avoids a large amount of storage for traces when the synthesis extracts less information than is generated by the monitor thread (e.g. reduction of variables across all cores or dynamic visualization with a sliding window).

- any slowdown in the synthesis engine or stream parser will back propagate to the monitor thread and cause a reduction of the sampling rate without any accumulation of records in transit.

Monitor thread

The monitor thread can be enabled on a per-simulation basis. When it is enabled, the following occurs:

- a separate thread of execution is created to run the monitor before the simulation starts;
- whenever the simulation starts or stops, the monitor is notified to start/stop collecting samples;
- while sample collection is active, the monitor periodically collects all *selected variables* and sends them to the monitor output channel in raw binary format.

If the output channel is blocked (for instance because its buffer is full) the monitor thread suspends without impacting the simulation.

Simulation variables and selections

The Microgrid simulator has been modified to expose many variables used internally by the simulation. Although these variables are internal, their values also describe the state of the simulated Microgrid. These variables include the cycle counter, thead and family allocations, arbitration delays on each arbitrated port, cache and memory information, etc.

The variables are placed in a namespace that reflect the internal structure of the simulator. The list of all available variable names depends on the Microgrid configuration and can be queried using either:

- `mgsim -n -D ....`
- `slr .... -Mv list`
- in the interactive simulator prompt, with the command "`samplevars`".

Individual variables can be referenced by their full name. Typically, groups of variables are selected instead, using a *pattern* with the same syntax as the Unix shell. For example, the pattern `*cpu*.threads.maxalloc` selects the high water mark for the number of allocated threads per core across all cores.

> **Note**
>
> In Unix shell patterns, `?` matches any single character, and `*` matches any sequence (including no character). Specific classes of characters are matched by `[...]`. See the `sh(1)` manual page for details.

Monitor configuration

`mgsim` command line	`slr` command line	Description
`-m`	`-M`	Enable the monitor thread.
`-o MonitorOutputFile=FILE`	`-Mo FILE`	Output sample records to `FILE` (should be a FIFO, defaults to `mgtrace.out`)
`-o MonitorMetadataFile=FILE`	`-Md FILE`	Output the trace metadata to `FILE` (defaults to `mgtrace.md`)
`-o MonitorSampleDelay=SECS`	`-Mr SECS`	Wait `SECS` seconds between each sample (defaults to 0.001s).

`-o MonitorSampleVariables=PATTERNS`	`-Mv PATTERNS`	Select all variables whose names match any of the `PATTERNS` for monitoring.
`-D`	`-Mv list`	List all available simulation variables.

The default selection pattern is `*.cpu*.pipeline.execute.op,*.cpu*.pipeline.execute.flop`.

Irrespective of the selection, the special variable `kernel.cycle` (cycle counter) is always sampled two times in each record, one before and one after all other variables are sampled. This allows to estimate "how long" the sampling itself has taken in terms of simulation time.

Monitor output

The monitor thread outputs data to two separate streams: the *metadata* and the *record* streams.

The metadata is output once before the simulation starts. It consists of a textual output which describes the simulation configuration as well as the list of variables being sampled. Its format is described in Metadata format below.

The records are output periodically during the simulation. All records have the same fixed with; they consist of the following data in their native machine representation concatenated together:

a Unix timestamp (absolute time) at the point the sampling of this record started;
a Unix timestamp at the point the sampling of this record ended;

the cycle counter (simulated time) sampled before all other variables;

each sampled variable;

the cycle counter sampled after all other variables.

The width (in bytes) and the type (integer/float) of each variable in a record is indicated in the metadata.

Examples

```
$ slr -M 10threads
```

Execute the program `10threads` with monitoring. The metadata is output to `mgtrace.md` and the records to `mgtrace.out`. Each sample contains the cycle counter and the counts of all instructions executed and floating point operations issued for all cores.

```
$ slr -Mv '*.threads.maxalloc' fibrec
```

Execute the program `fibrec` with monitoring (`-Mv` implies `-M`). Each sample contains the cycle counter and the high water mark of core thread allocations per core for all cores.

```
$ mkfifo rec-out; slr -Mo rec-out fibrec
```

Execute the program `fibrec` with monitoring. The records are output to a FIFO. The monitor is suspended until another process reads from the FIFO, but the simulation is unaffected.

Trace parsing and result synthesis

A helper program `readtrace` is provided to read the binary record stream, synthesize results and produce textual output stream.

This program operates as follows:

the metadata is read to obtain the width of records and the type and width of each sampled simulation variable;
a *synthesis engine code* (written in Python, user-specified) is loaded and byte-compiled;

the record stream is opened, each record is read in turn, transformed into a Python tuple and provided to the synthesis engine;

at the end of the input, the program reports statistics about the processing speed.

The synthesis engine is in charge of extracting the variables of interest and synthetising abstract results from them before performing a textual output.

Synthesis engines

Synthesis engines are specified using a single Python source code file. Engines should be structured as follows:

```
<code executed once>
for i in input:
     <code executed for each record>
     tabulate(variables...)
```

The variable `input` is a generator predefined by the stream parser that produces the data records as Python tuples. The function `tabulate()` formats its arguments into columns.

To access individual counters in each record, the namespace from the metadata is exposed in Python as variables that hold indices into the record tuple. For example, the following engine extracts the wall clock time and the simulated time as two columns:

```
for i in input:
     tabulate(i[wallclock.sec] + .000001 * i[wallclock.usec],
              i[kernel.cycle])
```

The following entities are predefined:

`input` The generator of input records.

`zerorec` A tuple of the same width as records, filled with zeros.

`mgconf` A namespace populated with the Microgrid configuration parameters. The list of all configuration names can be obtained using `print dir(mgconf)`.

`select(PATTERN)` Function: builds a list with the indices of all simulation varibles whose names match `PATTERN`. This can then be used e.g. with a comprehension to perform reductions. For example:

```
sel = select('*.threads.maxalloc')
for i in input:
    maxall = sum((i[x] for x in sel))
    tabulate(i[kernel.cycle], maxall)
```

> **Note**
>
> The cost of `select()` may be high. When possible, save the result in the initialization code at the start.

`window(N)` Constructor: defines a circular buffer of `N` entries. Entries can be inserted using the method `append()`. This can be used e.g. to perform sliding averages as follows:

```
acw = window(100)
sel = select('*.threads.curalloc')
for i in input:
    nactive = sum((1 for x in sel if i[x] != 0))
    acw.append(nactive)

    avg_nactive = sum(acw) / len(acw)
    tabulate(i[kernel.cycle], avg_nactive)
```

When no engine is specified, the following default code is used:

```
wcstart = None

for i in input:
  # wall clock time at start of sample
  wt1 = i[wallclock.sec] + .000001*i[wallclock.usec]
  # wall clock time at end of sample
  wt2 = i[wallclock.sec_] + .000001*i[wallclock.usec_]

  # simulated time at start of sample
  st1 = i[kernel.cycle]
  # simulated time at end of sample
  st2 = i[kernel.cycle_]

  # compute "middle" values:
  wc = (wt1+wt2)/2.
  st = (st1+st2)/2.

  # keep track of the initial wall clock time
  if wcstart is None:
     wcstart = wc-0.000001

  # prepend the times to the record data
  tabulate([wc-wcstart, st] + list(i))
```

Synthesis recipes

The following snippets can be used to synthetise results not directly expressed in simulation variables:

- ```
  iprev = zerorec
  for i in input:
        <compute, tabulate>
        iprev = i
  ```

  Save each record for the next iteration in `iprev`. This allows to perform differentials.

- ```
  iprev = zerorec
  sel = select('*.execute.op')
  for i in input:
      ops = sum((i[x]-iprev[x] for x in sel))
      tabulate(i[kernel.cycle], ops)
      iprev = i
  ```

  Count the number of instructions executed by all cores between each sample point.

- ```
  iprev = zerorec
  sel = select('*.execute.op')
  for i in input:
      activecores = sum((1 for x in sel if (i[x]-iprev[x]) != 0))
      tabulate(i[kernel.cycle], activecores)
      iprev = i
  ```

  Compute the number of active cores at each sampling point, by considering which cores have actually executed instructions between each sample.

- ```
  iprev = zerorec
  stprev = 0
  sel = select('*.execute.op')
  r = mgconf.masterfreq / mgconf.corefreq
  for i in input:
      ops = sum((i[x]-iprev[x] for x in sel))
      activecores = sum((1 for x in sel if (i[x]-iprev[x]) != 0))
      st = i[kernel.cycle] / r
      slots = activecores * (st - stprev)
      if slots == 0: continue
      pl_eff = ops / float(slots)
      tabulate(i[kernel.cycle], pl_eff)
      iprev = i
      stprev = st
  ```

  Compute the pipeline efficiency at each sample, as the ratio between the number of instructions executed and the number of pipeline slots available between each sample.

- ```
  iprev = zerorec
  stprev = 0
  sel = select('*.execute.op')
  effw = window(100)
  r = mgconf.masterfreq / mgconf.corefreq
  for i in inputs:
      ops = sum((i[x]-iprev[x] for x in sel))
      activecores = sum((1 for x in sel if (i[x]-iprev[x]) != 0))
      st = i[kernel.cycle] / r
      slots = activecores * (st - stprev)
      if slots == 0: continue
      effw.append(ops / float(slots))
      tabulate(i[kernel.cycle], sum(effw)/len(effw))
      iprev = i
      stprev = st
  ```

  Same as above, but the pipeline efficiency is averaged across a sliding window of 100 samples.

Producing diagrams

The synthesis engine emits its output as pure tabulated numerical output. The exploitation of this data by plotting tools is left as an exercise to the reader.

Future work

This framework expresses a loose coupling between the key components: the format of the synthesis output is dependent on the code of the synthesis engine; and the validity of a synthesis engine for
  ```

a given simulation trace is dependent on which variables it uses and which variables have been selected by monitoring in the simulator.

At the time of this writing, the three aspects (variable selection, choice of synthesis engine, data exploitation) must be configured separately. Care is thus required to ensure that they are compatible, otherwise errors can occur or results can become meaningless.

As future research it may be interesting to design a front-end tool, possibly in the form of a graphical user interface, that drives both the simulation and the synthesis.

Metadata format

The record metadata consists of one or more sections, each using the following syntax:

`#` *identifier* `:` [*text-words...*]

[*text-lines...*]

That is, a section starts with `#`, then an identifier which uniquely identifies the section, then a colon, then an optional sequence of words separated by spaces, then a newline character, then an optional sequence of text lines each terminated by a newline character.

The following sections are mandatory:

- "`varinfo: N`": indicates that each record in the record stream contains information for `N` simulation variables. The section then contains `N` additional lines of text that describe the width and type of each variable;
- "`recwidth: N`": indicates that each record is `N` bytes wide. No additional line of text is present;
- "`tv_sizes: A B C`": indicates that `struct timeval` (for Unix timestamps) has a width of `C` bytes, that its field `tv_sec` is `A` bytes wide and that its field `tv_usec` is `B` bytes wide. No additional line of text is present.

The following sections are optional:

- "`date: DATETIME`": indicates when the simulation started;
- "`generator: STRING`": identifies the simulator;
- "`host: STRING`": indicates on which host the simulation was run;
- "`program:`": indicates the name of the program executed on the Microgrid. A single additional line of text contains the program name.
- "`inputs: N`": indicates that `N` input files where loaded in the simulated memory prior to program startup. The section then contains `N` additional lines of text each containing the name of an input file.
- `confwords: N`: describes the Microgrid configuration. The section then contains one additional line of text, within which `N` configuration words are expressed.

C SL Library: performance counters (TR)

SL Library: performance counters

Key: sl8
Date: 2009-11-18
Status: Draft
Author: Raphael 'kena' Poss
Author: Andrei Matei
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/sl8.txt
Version: sl8.txt 3088 2009-11-18 13:54:04Z kena

Abstract This note documents an interface to access low-level global performance counters from SL programs.

Introduction

We need performance counters for benchmarking.

Benchmarks can have the following structure:

```
declare samples[N];
do I .. N: /* outer loop */
   before = get_counter();
   do_work();
   samples[i] = get_counter() - before;
do I .. N:
   report(samples[i])
```

This approach is valid for time, global cache activity, global number of instructions, etc. We can thus try to capture this approach in an API in a way *friendly to the later addition of new counters.*

Overview of the low level interface

For accessing absolute time with a fine granularity, C99 defines the `clock()` function as an "abstract measurement of execution time from an unspecified start point" and the symbolic constant `CLOCKS_PER_SEC`, expected to provide sub-second precision. This is also supported in SL, declared in `<ctime.h>`.

Note

On the Microgrid simulator C's `clock()` is currently implemented to return the cycle counter and `CLOCKS_PER_SEC` is set to an arbitrary value of 10e9 to reflect the estimated 1GHz clock rate. Also, as an extension the C function definition is complemented by a macro definition of the same name which expands to an efficient sampling of the cycle counter instead of performing a function call. (The C function can still be reached by undefining the macro and/or parenthesizing the name `clock` appropriately.)

Note

The `get_cycles()` API "historically" available in SL (dating from before the authoring of this note) is now deprecated in favor of C's `clock()` which has equivalent semantics and a well-defined standard interface.

For accessing additional counters, the following additional API is available from `<svp/perf.h>`:

- `mtperf_sample`, which efficiently samples all the available counters and stores them in an array;

- `mtperf_sample1`, which efficiently returns the current value of one of the counters;
- `mtperf_report_diffs`, which takes two arrays previously filled with `mtperf_sample` and reports their difference in a consistent format on the console output.

Low level interface details

The definitions discussed below are available after including `<svp/perf.h>`.

Data type

All counters have the data type `counter_t`. This is an integer type.

Available counters

The number of available counters (for the purpose of allocating arrays of `counter_t`) is available via the compile-time constant `MTPERF_NCOUNTERS`.

Each counter is identified by a preprocessor macro name:

- the preprocessor macor is defined only if the corresponding counter is available;
- the macro is used in conjunction with `mtperf_sample` and `mtperf_sample1` as discussed below.

The following table lists the counters available at the time of this writing; this list is non exhaustive and may/will be extended in the future:

Macro name	Description
`MTPERF_CLOCKS`	Number of clock ticks. Equivalent to C's clock().
`MTPERF_EXECUTED_INSNS`	Total number of completed instructions.
`MTPERF_ISSUED_FP_INSNS`	Total number of issued floating-point operations.

Sampling a single counter

The following function-like service is available:

```
counter_t  mtperf_sample1(int counter);
```

A call to `mtperf_sample1` returns the current value for the specified counter (one of the symbolic names defined above).

For example:

```
counter_t c1 = mtperf_sample1(MTPERF_CLOCKS);
do_something();
counter_t c2 = mtperf_sample1(MTPERF_CLOCKS);

printf("clocks: %d\n", c2 - c1);
```

Sampling all counters

The following function-like service is available:

```
void mtperf_sample(counter_t *counters);
```

A call to `mtperf_sample` will fill the array at the specified address with the current values of existing counters.

If the array has a size smaller than `MTPERF_NCOUNTERS`, the behavior is undefined.

After the `mtperf_sample` call completes, each individual counter can be accessed by indexing the array by the name of the counter.

For example:

```
counter_t ct[2][MTPERF_NCOUNTERS];

mtperf_sample(ct[0]);
do_something();
mtperf_sample(ct[1]);

printf("clocks: %d\n", ct[1][MTPERF_CLOCKS] - ct[0][MTPERF_CLOCKS]);
```

The `mtperf_sample` service is intended to be more efficient than using `mtperf_sample1` repeatedly to sample every counter, although one use of `mtperf_sample` is not necessarily as efficient as one use `mtperf_sample1` to sample a single counter.

Reporting differences

The following function is declared:

```
void mtperf_report_diffs(counter_t *before, counter_t *after, int flags);
```

When called, this service will print all the counter differences between the second and the first array. The extra argument `flags` determines the format of the output as a bitwise OR of the following flags:

- `REPORT_RAW`: the differences are output standalone, one per line of output.
- `REPORT_CSV`: the differences are output as a row of values suitable for input in a spreadsheet. When using this format, the additional flags can be ORed together:
  - `CSV_SEP(C)`: use `C` as a separator between columns (default is a comma)
  - `CSV_INCLUDE_HEADER`: also print the counter names as column headers
- `REPORT_FIBRE`: the differences are output as a Fibre array.
- `REPORT_STREAM(N)`: the report is printed on the stream `N` (1 = standard output, 2 = standard error, default is 1).

New formats may be added in the future.

Overview of the higher-level interface

For accessing additional counters, the following additional API is available from `<svp/perf.h>`:

- `struct s_interval`, item which can be manipulated by the functions below;
- `mtperf_start_interval`, which starts a measurement "section" and optionally tags it with a string and/or a number;
- `mtperf_finish_interval`, which ends the matching interval;
- `mtperf_report_intervals`, which takes an array of intervals and reports the metrics in a consistent format;
- `mtperf_alloc_intervals`, which allocates an array of type `struct s_interval`;
- `mtperf_free_intervals`, which deallocates an array of intervals;

Higher-level interface details

The following services are defined:

- `void mtperf_start_interval(struct s_interval *ivs, size_t p, int numtag, const char *strtag)`

  Begins a new interval at position `p` in the array pointed to by `ivs`. If `strtag` is non-null, it is used to annotate the interval textually. If `numtag` is positive or zero, it is used to annotate the interval numerically. (both `strtag` and `numtag` can be provided)

- `void mtperf_start_interval(struct s_interval *ivs, size_t p)`

  Ends the interval at position `p` in the array pointed to by `ivs`.

- `void mtperf_report_intervals(struct s_interval *ivs, size_t n, int flags)`

  Produces a textual report of the `n` first intervals in the array pointed to by `ivs`. The flags are defined in the same way as `mtperf_report_diffs` documented in Reporting differences above.

  In the special case of Fibre, the output is split in two sections:

  - in a first section delimited by the text "`### begin intervals`" and "`### end intervals`", the metrics are reported;
  - in a second section delimited by the text "`### begin descriptions`" and "`### end descriptions`", the interval names and counter names are reported.

- `struct s_interval *mtperf_alloc_intervals(size_t n)`

  Returns `calloc(n, sizeof(struct s_interval))`.

- `void mtperf_free_intervals(struct s_interval *ivs)`

  Performs `free(ivs)`.

- `void mtperf_empty_interval(struct s_interval *ivs, size_t p, int numtag, const char *strtag)`

  Clears the interval at position `p` in the array pointed to by `ivs`. Calling this function is not *required* on intervals allocated with `calloc` or `mtperf_alloc_intervals` above, but it allows to attach a tag to an interval otherwise remaining empty.

Example for the low level interface

The following program iterates a benchmark a number of times specified externally, and reports the results at the end:

```
#include <svp/perf.h>
#include <svp/slr.h>


slr_decl(slr_var(unsigned, L, "outer iteration count"));

sl_def(t_main, void)
{
   unsigned i;

   // get iteration count from environment
   unsigned L = slr_get(L)[0];

   // allocate an array of arrays of counters
```

```
   counter_t  counters[L+1][MTPERF_NCOUNTERS];

   // perfom the benchmark, and sample counters at
   // each outer iteration
   mtperf_sample(counters[0]);
   for (i = 1; i <= L; ++i)
   {
      do_benchmark();
      mtperf_sample(counters[i]);
   }

   // after benchmark completes, report counts
   for (i = 1; i <= L; ++i)
      mtperf_report_diffs(counters[i-1], counters[i], 0);
}
sl_enddef
```

Example for the high level interface

The following program iterates a benchmark 3 times, and reports the results at the end:

```
#include <svp/perf.h>

sl_def(t_main, void)
{
   unsigned i;
   size_t p = 0;

   // some arbitrary iteration count
   unsigned L = 3;

   // allocate L intervals
   struct s_interval *ivs = mtperf_alloc_intervals(L + 2);

   // benchmark initialization
   mtperf_start_interval(ivs, p, -1, "init");
   do_initialize();
   mtperf_finish_interval(ivs, p++);

   // perfom the benchmark, and sample counters at
   // each outer iteration
   for (i = 0; i < L; ++i)
   {
      mtperf_start_interval(ivs, p, i, "work");
      do_benchmark();
      mtperf_finish_interval(ivs, p++);
   }

   // benchmark teardown
   mtperf_start_interval(ivs, p, -1, "teardown");
   do_teardown();
   mtperf_finish_interval(ivs, p++);

   // after benchmark completes, report counts in Fibre format
```

```
    mtperf_report_intervals(ivs, p, REPORT_FIBRE|FIBRE_PAD(7));
}
sl_enddef
```

A possible output for this program would be:

```
### begin intervals
[1,3: 55 7 0]
[1,3: 55 7 0]
[1,3: 19 7 0]
[1,3: 19 7 0]
[1,3: 12 7 0]
### end intervals
### begin descriptions
[1,7: "init" "0 work" "1 work" "2 work" "teardown" "" ""]
[1,7: "clocks" "n_exec_insns" "n_issued_flops" "" "" "" ""]
### end descriptions
```

D Generalized I/O events for the Microgrid (TR)

Generalized I/O events for the Microgrid *[now partially superseded]*

Key: svp21
Author: Joe Masters
Author: Mike Lankamp
Author: Raphael "kena" Poss
Date: 2008-04-14
Status: Draft
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/svp21.txt
Version: svp21.txt 1972 2009-04-17 00:02:45Z kena

Abstract In this note we propose two approaches to implement I/O event signalling on the Microgrid.

Introduction

We connect the Microgrid to a generic I/O device[1] which delivers interrupts to the Microgrid. The hardware interface between the device and the Microgrid is present on specific cores hereafter named "I/O cores".[2] We focus here on the way the programs running on the Microgrid can wait on external I/O *events*. The way programs can read and write *data* to devices is outside of the scope of this document.[3]

We propose two approaches which can be used in conjunction, although it appears after the initial research phase that only the first approach can be successfully implemented.

We assume the availability of exclusive places (cf. **[svp24]**_).

Common concepts

On the I/O core we define a subset of general-purpose integer registers to be *I/O registers*. Each of these registers is connected to an external interrupt channel. The state of these registers is dependent on threads waiting on them and interrupt delivery as follows:

- an I/O register is *empty* when no threads are waiting on it. If an interrupt is delivered for the channel while the register is empty, the interrupt is stalled (either buffered, queued or latched). From that point the interrupt is said to be "pending".
- like for all registers on the Microgrid architecture, an I/O register is *waiting* when at least one thread is waiting on it:
  - when one or more threads start to wait on an I/O register and no interrupt is currently pending for the channel, the threads are suspended and the register becomes "waiting";

[1] this can be: a serial interface, a network interface, etc. Genericity can be achieved by connecting a device through which multiple channels can be multiplexed.

[2] whether I/O cores should be specialized members of the Microgrid or whether I/O can happen symmetrically on all cores is a matter of configuration and manufacturing costs; it is not discussed here.

[3] we assume here that data exchange and data acknowledgement can be implemented via traditional means (memory-mapped, special ports, special address space, etc). This will be discussed in a separate note.

  - when one or more threads are suspended and the register is waiting, and then an interrupt is delivered, the threads currently waiting on the register are woken up;
  - when one or more threads start to wait on an I/O register and an interrupt is currently pending for the channel, the threads are not suspended.

- as a difference with the usual register behavior in the Microgrid, we propose that a waiting register is marked "empty" when an interrupt is delivered.

Continuation-based I/O

In this approach each I/O register is waited upon at most one *server* thread whose role is to wait for events and deliver them to *clients*.

The overall process goes as follows:

- client applications *register* as *listeners* to I/O events;
- the server threads wait for events and *signals* the client applications when the event is delivered using continuation families (cf [svp26]_).

In this approach, when looking at the concurrency tree defined by the parent-child relationship between families, the families of threads created to handle incoming events have the server threads as ancestor. There is no parent-child link between the other families in the application process and the families executed to handle events.

Overview diagram

The following figure illustrates this approach.

Common items

This mechanism relies on a shared data structure for each I/O event, representing a queue where each node contains create parameters for a continuation family: pointer to code, optionally shared/global arguments, optionally index range parameters.

All accesses to this shared data structure as described below is done on an exclusive place mutually known by the clients and the server threads. When handling multiple I/O event channels, each event channel is associated to one such data structure, which itself must be accessed via at most one exclusive place; however data structures for distinct I/O channels can be accessed using distinct exclusive places to achieve additional concurrency.

Registration of a client application as a listener

We propose two registration mechanisms:

- for *asynchronous event delivery*, a thread in a client application can queue an *event handling continuation* in the form of a pointer to code, data and protection domain to be created as a continuation family by the server when the event happens; the client thread can then concurrently continue to execute and terminate after it has registered its continuation and before the event is signalled. This mechanism is similar to registering a callback on a legacy system.
- for *synchronous event delivery*, a thread in a client application creates a sear[4] which queues to the server, as a event handling continuation, a thread code that kill the sear; then the sear suspends, so that its parent waiting for termination of the sear suspends as well. When the event is delivered the continuation kills the sear, allowing the client thread to resume.

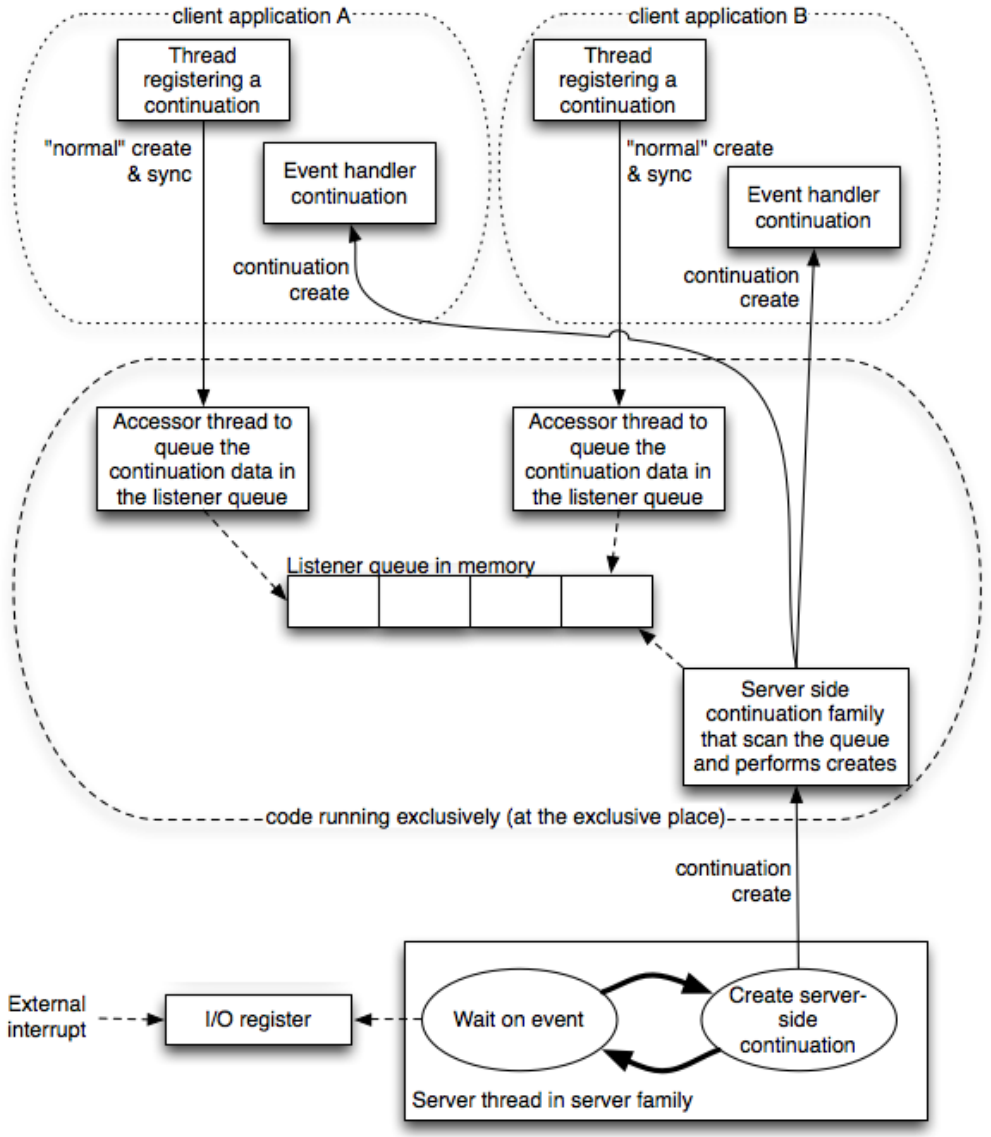


Figure 9: Overview of the continuation-based I/O event handling mechanism

Waking up clients by the server thread

From the server thread perspective, we describe the behavior as the following sequential loop:

> the server thread waits on the I/O register corresponding to the event being listened to by the clients;
> when the interrupt is delivered, the server thread wakes up, then creates a *server-side infinite continuation family* at the exclusive place;
>
> then because this child family is a continuation, the server thread does not need to wait for its termination and can start waiting for the next event.

The server-side infinite continuation family scans the listener queue for the event, and then in each thread of this family the individual listeners are handled as follows: a *client-side continuation family* is created using the pointer to code, data and protection domain registered by the client in the queue.

Cooperative I/O

> **Warning**
>
> This approach is described here for "historical" purposes, but seems to be conceptually faulty.

In this approach each I/O register is waited upon by one or more I/O *peer threads* synchronously and simultaneously waited upon by multiple client applications: the client application threads are waiting for the I/O peer threads to terminate using the regular SVP "synchronization on termination" mechanism.

The overall process goes as follows:

- each client application interested in an event creates a driver family of one thread which then waits for events by creating short-lived threads at the I/O core;
- when an interrupt is delivered, all the waiting threads are resumed, terminate, the driver families resume and then handle the event and return the requested information to the client application.

In this approach, when looking at the concurrency tree defined by the parent-child relationship between families, the families of threads created to handle incoming events have another thread in the application process as an ancestor.

Function of a single I/O thread

All threads waiting on an I/O channel are cooperatively responsible for buffering input events. This is because some threads will see events delivered in which their "owner" client application is not interested.

As a result, for each event delivered a waiting thread needs to perform the following:

1. check an event buffer and see if there is an event already delivered;
2. if there is an event already there and the event is "interesting" to the waiting thread, simply return it to the parent client application and terminate; or
3. wait on the next event;
4. put the new incoming event on the buffer; if it is "interesting" to the waiting thread return it to the parent client application.

[4] as defined in [svp20]: a family that indefinitely suspends, e.g. a family of one thread waiting on a shared thread argument that is not provided by the parent.

Issues remaining to be resolved

This second approach requires exclusive access to the buffer; care must be taken to ensure both proper atomic access to the buffer data, and proper concurrent access to the I/O register. After a thorough analysis, a workable, efficient and correct exclusion protocol has not been found yet.

Mapping between interrupts and I/O threads

The two approaches described above assume that the I/O registers are visible in the local thread context of the I/O threads. For this purpose we propose the following two mapping mechanisms:

- one mechanism where I/O registers are mapped *automatically* in the I/O threads as global thread parameters; this uses a similar mechanism as the mapping of global thread arguments between a parent thread and a child family, as described below;
- another mechanism where an *extra instruction* in the ISA is provided to threads to establish a connection between an arbitrary interrupt line and an arbitrary register in the register file; this requires an extra instruction and an on-chip lookup table.

Automatic mapping

An I/O family is marked as such by a flag set by its parent thread at the point of create. When the create request is handled in the I/O core, the special mapping occurs when the flag is set.

The mapping connects a subset of the thread register window of the newly created thread to some subset of the I/O registers. For example, we can define that the N first global registers for the newly created thread are mapped to the N I/O registers connected on the core to interrupts (N <= 31).

This requires a specialized on-chip thread creation process for I/O threads.

Explicit mapping

In this scheme the threads are allocated "as usual" on the I/O core (using the same thread creation process as on other SVP cores). While a thread is running, it can execute an "I/O map" instruction.

This instruction takes as input an arbitrary register specifier taken from the local register window of the thread and an arbitrary I/O channel identifier. It updates a lookup table on the core that connects the absolute register offset of the specified register in the register file to the interrupt line corresponding to the specified I/O channel.

After the the thread has established the mapping it can enter a loop of waiting on the event register and handling the delivery to client applications.

By software convention we can define that the driver code will establish a one-to-one relationship between registers and interrupt lines.

Glossary of terms for I/O events

client application Collection of related families that are "interested" in receiving I/O events.

client-side continuation family / event handling continuation Family created by the server thread using the "continuation create" mechanism when an I/O event occurs, using continuation information previously registered by a client application.

I/O core Core where some integer registers are configured as I/O registers.

I/O register Register whose state is linked both to the threads waiting on it and to the state of an external I/O interrupt line.

server thread / server family Family of one thread running hardware driver code on an I/O core.

server-side infinite continuation family Family of thread created as a continuation by the server thread to concurrently create client-side handler continuations while the server thread can start waiting for the next event.

asynchronous event delivery Occurs when the handling of an event in an application is done independently from the thread that registered the continuation handler for the event.

synchronous event delivery Occurs when an application thread suspends until the event it is listening to is delivered.

E Towards a Microgrid Hardware I/O Mechanism (TR)

Towards a Microgrid Hardware I/O Mechanism *[now partially superseded]*

Key: svp31
Author: Michael A. Hicks
Author: Chris Jesshope
Author: Mike Lankamp
Author: Raphael "kena" Poss
Author: Michiel van Tol
Date: 2009-12-05
Status: Draft
Replaces: svp22
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/svp31.txt
Version: svp31.txt 3927 2009-12-17 00:32:59Z kena

Introduction

In this document we propose a general architecture-level schematic for external device I/O on the Microgrid, conforming with the direction set out in [svp23]_. It is designed to bypass the delay of main memory and enhance throughput for on-chip parallel execution.

The proposed implementation uses dedicated 'I/O Cores', to which extremely lightweight device driver threads are delegated, to receive interrupts and communicate with devices over a high speed bus. The behaviour of components beyond this I/O bus is not detailed in this document. A bus interface component exists on each I/O core and serves to communicate control through registers and data directly to the on-chip COMA.

This document does not focus directly on the software level I/O protocols discussed in [svp21]_, but is intended to be relatively compatible with those methods. Note [svp22]_ is made obsolete by the outline proposed here.

Architecture Level Description

In this model (*Figure 1*), the Microgrid communicates with I/O devices through a high speed packet bus, proposed here as a *'HyperTransport-like bus'*. This communication can only take place on special hardware *I/O cores*. These cores implement a special register file which contains designated *control registers* in each thread context. These registers are used to control the *Bus Interface*, in the form of a channel number and data packet pointer and size. In addition, the properties of synchronising registers are exploited to enable threads to wait on interrupt events delivered by the bus interface. In turn, the bus interface handles how such data packets and events will be composed onto the bus.

One or more external devices can be connected to this high speed packet bus or, if required, a legacy interface can be connected which allows commodity buses, such as the commonplace PCI Express and USB, to be used with the Microgrid through a single standard interface.

The proposed implementation does not preclude the use of a DMA controller for direct memory transfers. This is useful in such applications as, for instance, network routing, where inter-device transfers are necessary with minimal processing of the I/O data in between.

Components Abstract

I/O Cores The I/O Core is a simplified version of a standard microthreaded core on a Microgrid. It contains no floating point logic or floating point register file. Instead, it is equipped with a bus interface to the high speed I/O bus and an I/O register file which possesses special synchronisation and behavioural semantics.

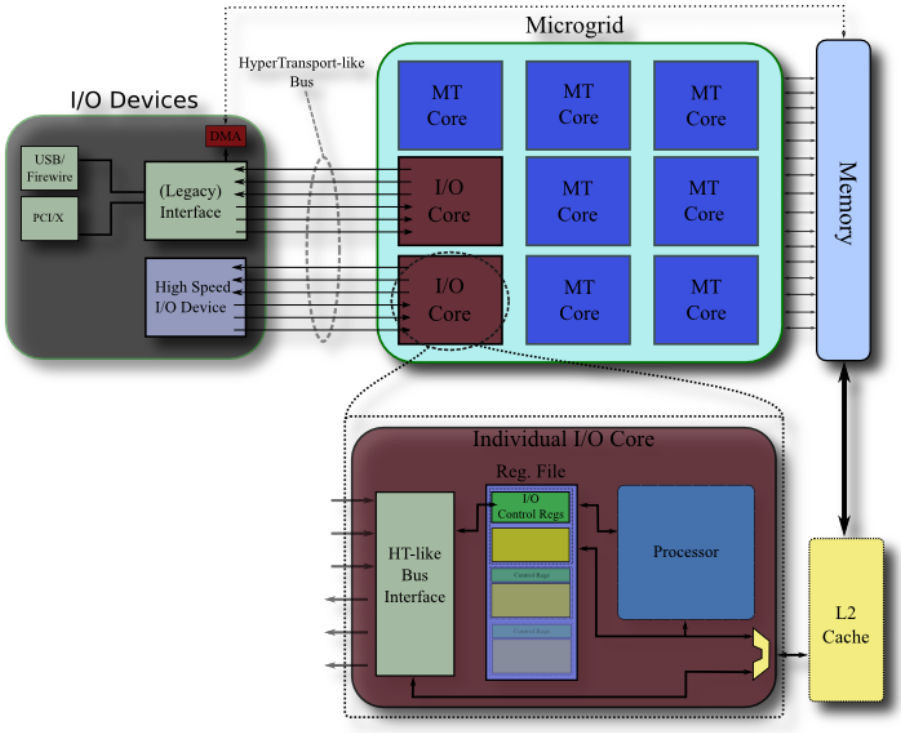


Figure 10: Figure 1: A schematic view of the proposed I/O architecture and role of I/O Cores in the Microgrid

HyperTransport-Like Bus A packet bus for high speed communication with devices. I/O packets would consist of a control header (including device/channel information) and a data payload. Further detail below.

Bus Interface A hardware interface device to the aforementioned bus. Has the function of composing/decomposing fine grained bus events and messages, and translating them into register file events and 'direct-to-cache' transfers. Each I/O core will contain a dedicated bus interface. Further detail in next section.

I/O Register File Every thread context on an I/O core contains a set of special registers which are used to communicate with the bus interface. These registers, essentially semantic extensions of synchronising registers, are used for both receiving I/O interrupt events (by waiting on a read to the register) and for initiating writes to an I/O device, by providing an I/O channel identifier, data pointer, and transfer size.

I/O Devices I/O devices can potentially be interfaced with the high speed I/O bus in a number of ways. Some devices, for instance HyperTransport devices, may be directly coupled with the bus. Others, legacy devices for instance, may require additional interface logic to mediate between differing device protocols. With HyperTransport as an example, this interface structure is already present in existing technology, with for instance AMD using HyperTransport to connect to traditional northbridge devices.

HT-Like Bus and Interface

HyperTransport:

A HyperTransport-like bus and protocol is proposed for communication in this model of Microgrid I/O. HyperTransport is a serial (or parallel, depending on configuration) bus specification for

high speed communication between devices. It is extremely well suited for the Microgrid because it is relatively simple, yet the specification is highly flexible. The method of communication specified in the HyperTransport whitepaper is packet based; one control packet and a variable length data payload. This fits well with a channel+data based model proposed for the Microgrid.

From the standpoint of the basic model, HyperTransport provides a suitable technology ceiling from which a feasible I/O bandwidth can be modelled and assumed. *Table 1* shows the current HyperTransport bandwidths and how they have scaled through time. It is safe to assume that the next version will again offer technologically improved bandwidth.

The term 'HT-like' is used since it may be desirable to make some departures from the official HyperTransport specification (or indeed not specify the details at all) and simply use it for feasible data-rate estimates and an approximate communication model (and, not least, to indicate easy integration with existing devices for the accelerator notion of a Microgrid).

HT Version	**Year**	**Max. Frequency**	**Max. Width**	**Max. Aggregate Bandwidth**	**Max. Bandwidth unidirectional**
1.0	2001	800 MHz	32-bit	12.8 GB/s	6.4 GB/s
2.0	2004	1.4 GHz	32-bit	22.4 GB/s	11.2 GB/s
3.0	2006	2.6 GHz	32-bit	41.6 GB/s	20.8 GB/s
3.1	2008	3.2 GHz	32-bit	51.2 GB/s	25.6 GB/s

Table 1: Advances in HT bus performance through time (note, HT is DDR)

The additional advantage of choosing an existing bus technology is that it provides known pin (and silicon) requirements on a given chip which uses HT for I/O. *Table 2* shows the varying pin requirements for the different link widths of hyper transport. The trade-off between transfer rate and silicon budget remains an open question for the Microgrid.

Link Width (each way)	**2**	**4**	**8**	**16**	**32**
Data Pins	8	16	32	64	128
Clock Pins	4	4	4	8	16
Control Pins	4	4	4	4	4
V ldt	2	2	3	6	10
GND	4	6	10	19	37
PWROK	1	1	1	1	1
RESET#	1	1	1	1	1
Total Pins	24	34	55	103	197

Table 2: Pin out requirements for various HyperTransport widths

HyperTransport communications, officially, take the following packet form:

- A series of 32-bit words, irrespective of link width. Communications padded to always align to word boundaries
- First word in the packet is always a control word, containing length and address (I/O channel) information

Bus Interface:

This means that the existing HyperTransport specification fits very closely with the proposed model of I/O in the Microgrid. The job of the *HT-like Interface* is thus relatively simple:

1) Compose and decompose HT bus messages into the associated channel/address information
2) Write (and read) channel event messages into appropriate I/O control registers
3) Perform reads and writes from/to memory of the data payload in I/O events at a specified location

This approach will require, at the very least, that the simulator model the internal communication carried out by the I/O interface. This will include the increased contention for the local aspect of the memory system and unconventional behaviour of the register file at I/O cores (see below). The intention is that threads running on an I/O core will not communicate directly with any HT-like protocol, but only the channel and data interface discussed in the next section. The way in which the interface communicates data to the local memory store, be it through a separate interface or just the cores standard memory interface, is as yet undefined.

The requirements of the bus interface also stipulate that it must contain a rudimentary interrupt controller, in order to map interrupt events to driver threads and store memory I/O pointers. However, given the overall model described here, the required logic is relatively simple when compared to a full APIC used in existing chipsets and the absence of floating point logic relieves pressure on silicon usage.

The I/O Register File

The I/O register file is a special extension to the concept of synchronising registers in the Microgrid and SVP model. Each thread context on an I/O core will have access to the usual set of integer registers, but no floating point registers, and some additional special I/O registers. These I/O registers are used to control reads and writes to external I/O devices by means of the Bus Interface using a channel/device identifier and some data to transfer in memory.

Acquiring a Device Context

Or How to Communicate with a Particular Device

There are many potential ways in which to acquire an I/O thread context with which to communicate with a particular device, assuming that one I/O core gives access to more than one device (otherwise one simply must delegate to the appropriate core).
The first of which is to assume that every thread context (of a 'driver') on a given I/O core is identical. Then, a register (in the case of the adjacent list, register 0) would be written to with a device identifier which instructs the bus I/O interface with which device it must signal from this thread.
An alternative is to use the family's initial thread index value (specified at the point of family creation) to indicate which device channel is required. In this case, one register in the 'driver' context is spared.
Finally, and applicable to all of the special I/O registers, is the use of a fixed calling convention. Each special register could correspond to a fixed position parameter in the driver thread's context. This would alleviate the need to perform any explicitly initialisation upon driver thread execution.
The final decision will depend on whether a fixed mapping of register contexts to I/O devices is used (if this is the case, then the I/O device/channel must be made explicit at the point of driver thread creation).

The special I/O registers will consist of at least the following, for both reading and writing:

Channel/Device Identifier (see 'Acquiring a Device Context')

Read/Write Interrupt Synchroniser

Data Buffer Pointer

Transfer Size

The set of I/O registers above would occur at a fixed position in each I/O core's thread contexts. Each of these special registers would be visible (i.e. mapped) the I/O Bus Interface.

An I/O thread (low-level driver) should initialise in the following way:

Write Channel/Device Identifier into register

Write data read/write address (buffer pointer)

Write transfer size (buffer size)

Either perform a read from *or* a write to the 'Read/Write Interrupt Synchroniser'

The 'Read/Write Interrupt Synchroniser', used in point 3 in the list above, will use the existing synchronising register concept of the Microgrid, modified with some special behaviour for I/O communication:

Read from Synchroniser register: the driver thread suspends until the I/O Interface receives an interrupt from the channel specified in the channel register and has completed writing the data payload into the buffer specified by the data buffer pointer (and of maximum size specified in the transfer size register).

Write to Synchroniser register: indicates to the I/O interface that a write should be performed of the data specified in the data buffer pointer (and of size specified in the transfer size register). A subsequent read to the synchroniser register would suspend until this action has completed.

These additional I/O registers and their associated semantics would permit the software level protocols described in [svp21]_ to be used as a driver and application level model.

Summary and Ongoing Work

- The Microgrid will feature specialised I/O cores
- A fast HyperTransport-like bus connecting devices to each I/O Core (up to a potential 25.6 Gb/Sec in each direction)
- Each I/O core has a bus interface which mediates and composes/decomposes bus events at the hardware level, and performs direct transfers to COMA memory. This interface has the additional property of behaving as a rudimentary interrupt controller
- A modified register file with I/O registers which are used to control the I/O interface, and no floating point registers
- Architecture will largely support software protocol model described in [svp21]_, subject to potential modifications

Next Steps:
- Begin developing I/O API for software running on the Microgrid
- Prototype Low-level I/O model using PTL back-end to ensure feasibility of model

F Efficient heap allocation on shared memory SVP (TR)

Key: svp34
Date: 2010-04-14
Status: Draft
Author: Raphael 'kena' Poss
Author: Clemens Grelck
Version: svp34.txt 3640 2010-04-14 21:28:36Z kena
Source: svn+ssh://mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/svp34.txt

Abstract We describe a lock-free, TLS-based memory allocator.

Use case

We want to address the problem of frequent dynamic memory allocation by tasks in massively concurrent (virtual) shared memory systems, where the result of allocations is only shared with a few other tasks, where allocations may persist past the termination of a task, and where deallocation may be performed by a different task than the allocating task.

> **Note**
>
> Tasks are concurrent units of executions defined by the program; we assume that tasks are run (and scheduled cooperatively) within non-interruptible thread contexts, and that contexts are recycled when tasks terminated. This model is a subset of several current models for concurrency management, e.g. OpenMP tasks running on top of POSIX threads, SVP threads running on top of Microgrid thread contexts, Cilk tasks on top of Cilk worker threads.

In this context we identify the following aspects to be considered for scalability:

1. locking -- lock-free approaches should be sought to avoid contention;
2. locality -- administrative data structures should be distributed to avoid cache traffic for memory management alone;
3. space overhead -- administrative data overhead should be linear or constant in the number of threads, and fragmentation should be limited

And also next to scalability we need to diminish time overheads, in particular:

1. requests overheads -- the overhead for frequent, predictable and small-size requests should be low (comparable to stack-based allocation), while it may be larger for infrequent or unpredictable or large requests; this is in particular useful in Apple-CORE where objects (e.g. SAC array descriptors) may be allocated and deallocated often even by very small tasks;
2. setup overheads -- speculative execution of setup code to prepare administrative data for future requests should be minimized.

Conceptual overview

We assume that the shared address space is separated into two parts, the first part being shared among all threads and hosting some "external global heap" managed by some memory allocator not discussed here (possibly lock-based), and the second part being free for use by programs.

The allocator we propose uses this second part of the address space to serve "small" allocations (where each requested size is below an arbitrary threshold) using a lock-free strategy.

Any individual allocation requested for a larger size is delegated transparently to the external global heap allocator (which may be lock-based, depending on the implementation).

This request size threshold is configurable statically. Previous literature (refs?) suggest a threshold of 256 bytes, while the research in Apple-CORE using SAC suggests a larger threshold (e.g. 1/2Kbytes).

We then split the address space in thread-specific ranges. In each range we define a thread-local heap as follows.

Allocation requests within a thread are served by rounding the requested size to the nearest larger "block size", and searching for a free block. Free blocks are found in superblocks containing many contiguous blocks ("chunks") of the same size as an array. The superblocks are grouped in bins of a size class. When no free blocks are found in existing superblocks, new superblocks are allocated from storage and mapped within the thread-specific range.

Deallocation requests are served by tagging blocks as free. The actual free lists and superblock use count are updated either:

- by the thread performing the deallocation if it is also the owner of the thread-local heap, or
- by a garbage collection process later during allocation by the owner thread.

The allocator places new superblocks in the thread-local address space of individual threads, and each superblock is used exclusively by the thread-local heap corresponding to that address space. The administrative data for the thread-local heap can therefore found in constant time and trivially based on the thread identifier.

Also, entirely unused superblocks are released to the environment sometimes so that the storage can be reused by other thread heaps in their (different) address ranges.

Overview of the data structures

We call the *delegation threshold* the threshold on the size of memory allocation requests past which the proposed allocator delegates the request to the (external) global heap allocator.

We partition the part of the shared address space available to programs between all hardware threads. We call each region of the partition the *thread-specific address space*.

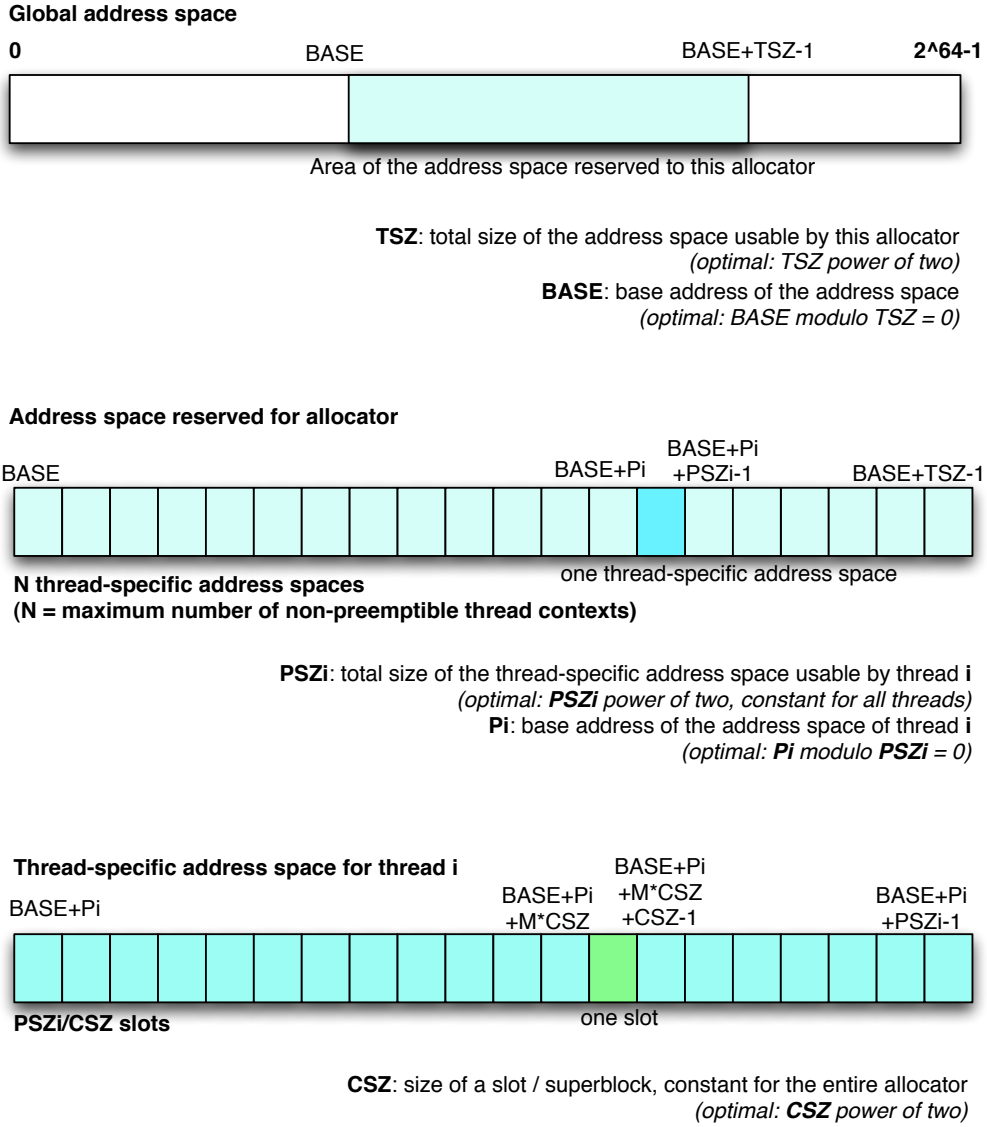
We then construct *thread heaps* inside the thread-specific space as follows.

First the thread-specific space is further divided into equally-sized address ranges called *slots*.

Note

At this point, the partition of the shared address space into thread-specific spaces, and the further division of thread-specific spaces into slots is entirely virtual, i.e. conceptually manipulates sets of *addresses* without requiring physical storage to be mapped to these addresses.

The address space partitioning is illustrated in the following figure:



Some notes about the selection of the constant *CSZ* is provided in Slot size and alignment below.

We then assume the implementability of the *mapping* of slots, the operation that requests storage from the execution environments and maps this storage (using virtual memory mapping) into slots. An *unmapping* operation is not necessary but can be used if present.

> **Note**
>
> On Unix the allocation of storage and mapping at a specific address can be achieved by means of the system call `mmap`; on the Microgrid, the mapping is implicit on first access).

At the beginning of the thread-specific space, some slots are mapped and reserved for a *control block* which holds metadata for the thread-specific allocator.

The control block contains:

- an array of *bins* for different *block sizes* below the delegation threshold. Each bin is either empty (0) or contains the start/last links to a circular doubly-linked list of some slot-sized *superblocks* described below. The number of bins and the block size for each bin is statically configured.[1]
- the start link of a singly linked list of *available* superblocks (defined below);

For example in the proposed implementation we use 5 bins with block sizes 16, 32, 64, 128, 256.

- a bit vector of slots/superblocks in the thread-specific space, where the state of each bit indicates whether the slot is currently mapped, i.e. whether the corresponding superblock is currently allocated;

- the number of currently allocated superblocks in this thread heap;

- the number of busy superblocks (see below) from the allocated set.

Superblocks have the same size as a slot and are allocated when needed by mapping storage onto slots. Each allocated (existing) superblock has a state; this state is deduced from the lists to which the superblock belong: the superblock is *available* if it belongs to the available list in the control block; it is *busy* if it belongs to the list of some bin. The mechanisms are defined so that any allocated superblock belongs to exactly one list.

Each superblock contains:

- the next link of the list of available superblocks, or the next link of the list of superblocks in the bin if the superblock is busy;

- if the superblock is busy:

  - the prev link of the list of superblocks in the bin.

  - the start link of a singly-linked list of *free* blocks within the superblock (defined below);

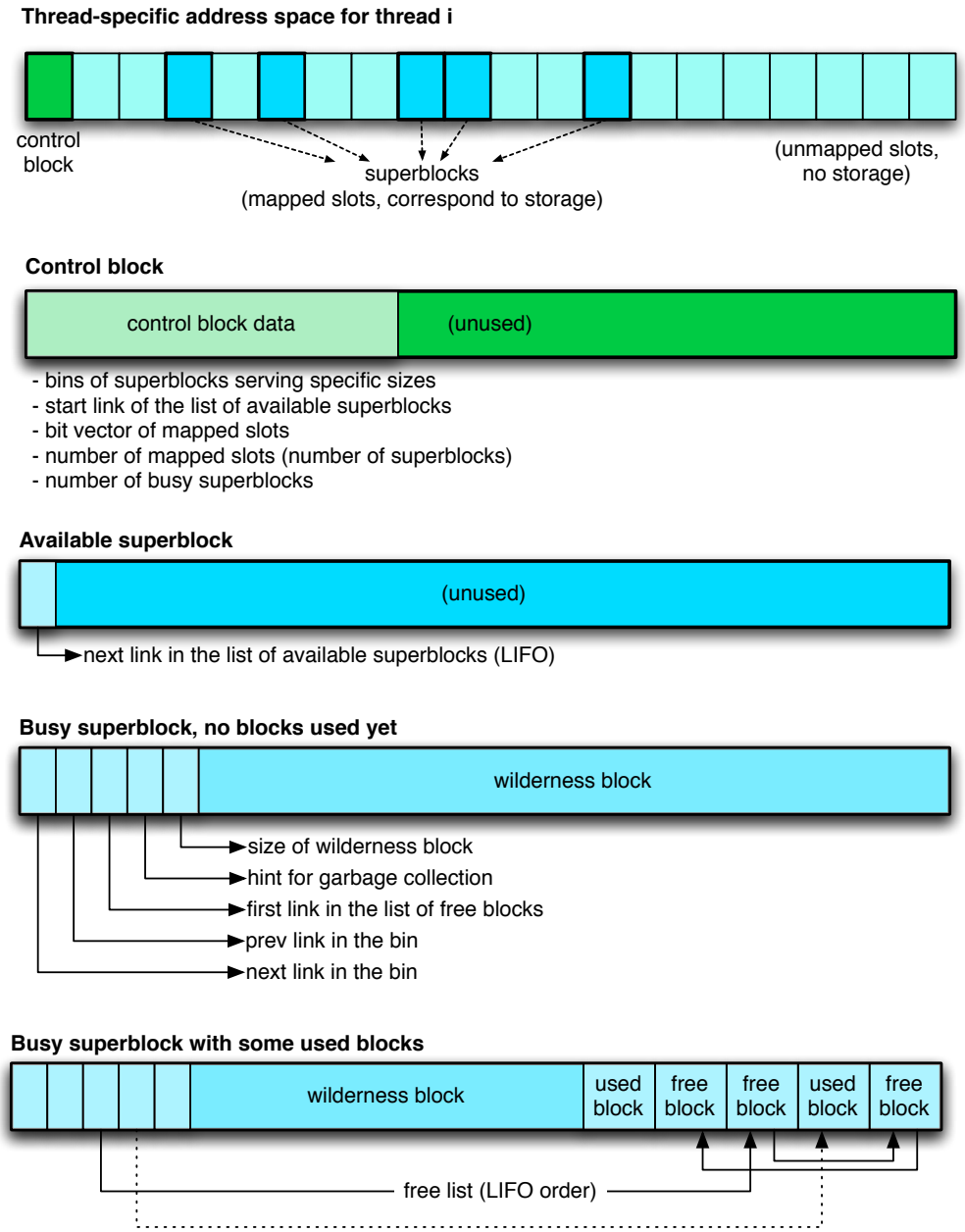
  - a *hint* to the garbage allocation process (below) to indicate where to resume garbage collection

  - a first *wilderness* block of variable size, with a size field at the beginning, initially sized to the remained of the superblock;

  - an array of equally-sized *blocks*, each of the superblock's block size,

- an variably-sized array of blocks whose end is aligned with the end of the superblock.

The structure of the control block and superblocks is illustrated as follows:



Each block in a superblock is composed of a *block tag* and a *payload*. The tag contains a value which indicates the *state* of the block. A block can be in two states, either *pending-free* (tag 0), or *non-pending* (tag > 0, either free or used but this distinction is not determined by the tag). When a block is free, the free link in the superblock's free list is stored in the payload.

Overview of the mechanisms

Overall optimization "tricks"

- when a block is freed by its own owner thread, the block is inserted into the free list, so that the next request for the same size can reuse the cache lines (improved locality)
- when a block is freed in a superblock, the superblock is placed at the head of the bin, so that the next request for that size can be services without finding a non-full superblock (improved locality, diminished overhead)
- when garbage collection occurs, it starts with the superblock at the head of the bin and within each superblock it scans circularly all blocks from a given "hint" position. A garbage collection that is part of an allocation stops as soon as the request can be serviced, and a "maintenance" collection stops after limited number of blocks/superblocks have been scanned. The point at

which collection stops is stored as a hint (for blocks in a superblock) or by updating the head of the bin (for superblocks in a thread heap).

- during garbage collection, freed blocks are placed in LIFO order in their respective free lists, or free blocks immediately succeeding the wilderness block are merged with it (improved locality, diminished fragmentation);
- when all blocks in a superblock become free, the superblock is moved to the available list so it can be reused by a different bin (diminished fragmentation)
- when the number of available superblocks exceeds a threshold, some superblocks are unmapped to release the corresponding storage
- because of the wilderness block, it is not necessary to construct an entire free list when a superblock is allocated (improved locality, decreased overhead).

Serving allocation requests (`tls_malloc`)

When allocation request is received for a size higher than the delegation threshold, it is delegated right away to the external global heap allocator.

Otherwise, the size is rounded up to the closest block size, and the corresponding bin is queried. The following then applies in order:

1. if the bin points to a non-full superblock (wilderness block not empty or free list not empty), then the superblock is used to serve the request directly (see below).
2. if the bin points to a full superblock (wilderness block empty + free list empty), then:
   a. the chain of superblocks for this bin is *garbage collected* (described below). This may cause some pending free blocks to be converted into free blocks, and may cause some entirely free superblocks to become available (removed from the bin, inserted in the available list).
   b. if the bin contains some non-empty superblock, this is used to serve the request (see below).
   c. otherwise (#b failed: all superblocks are confirmed full), allocation proceeds as per #3 below.
3. the bin needs a new superblock. The following happens:
   a. the allocated slot/superblock count in the control block is queried to check if there are any available slots in the thread-specific space (the number of available slots is the total number of slots in the thread-specific space minus the number of allocated superblocks indicated by the counter in the control block).
   b. if no slot is available, no more superblock can be constructed. The allocation request is delegated to the external global heap allocator.
   c. if some slot is available, the bitmap is searched for a free slot, and the mapping is attempted. If the mapping fails, the allocation request is delegated as per #b immediately above. If the mapping succeeds, the bitmap is updated, the slot/superblock count is increased, and the superblock is *initialized* (see below). Then allocation proceeds as per situation #1 above.

To serve an allocation request from a non-full superblock:

- if the free list is not empty:
  1. the first block from the free list is taken,

2. the block is removed from the free list,
3. the block is tagged non-pending
4. the address of the payload is returned as the result of the allocation.

- otherwise (the wilderness block is not empty)

1. a new block is carved at the end of the wilderness block
2. the size of the wilderness block is decreased
3. the block is tagged non-pending
4. the address of the payload is returned as the result of the allocation.

If a superblock becomes full as a result of an allocation, the bin pointer is updated to point to the next superblock in the chain (possibly the same superblock if there is only one, the chain is circular).

Deallocation requests (`tls_free`)

If the given address appears to lie outside of the thread-specific spaces, the deallocation request is delegated to the external global heap allocator.

Otherwise, the following is tested:

- if the given address lies outside of the thread-specific space of the thread performing the deallocation, then the tag address is deduced from the provided address and the block is tagged *pending free*.

> **Note**
>
> If the thread performing the deallocation is known to lie in the same *consistency domain* ([svp30]) as the thread owning the heap, then the tag can be updated by a local memory write. If the consistency domain may be disjoint, the write must be performed at the home place ([svp32]) of the tag, which is guaranteed to use the same consistency domain as the thread owning the heap.

  No further work is performed in this case, because the thread performing the deallocation is not the "owner" of the thread heap and may be running concurrently with some allocations.

- if the given address lies in the thread-specific space of the thread performing the deallocation, this means that this thread is not concurrently serving an allocation request. Then the following happens with no locking needed:

1. the tag address for the block is deduced from the provided address, and the block is tagged non-pending;
2. the block is inserted in the free list of its superblock;
3. If the superblock is non-empty, the bin pointer is updated to point at this superblock, because future requests can be honoured from this superblock
4. otherwise (superblock has become empty), then:
   a. the superblock is removed from its bin and inserted into the superblock available list in the control block,
   b. the count of busy superblocks in the control block is decremented.
   c. If an unmapping operation for slots is available in the implementation, and if the busy count becomes lower than some threshold (e.g. half the count of allocated superblocks), then some available superblocks are released (unmapped) to the environment.

Superblock initialization

When a fresh superblock is allocated in a slot for a given bin, the following need to be initialized:

- the wilderness block must be constructed to fill the entire superblock (size set to superblock size minus header size),
- the prev and next links must be updated to insert the superblock into the bin,
- the busy superblock count in the control block must be incremented.

Garbage collection of superblocks in a bin

Garbage collection iterates over some superblocks in a bin (number TBD). For each scanned superblock, some blocks in the superblock are scanned starting from the "hint" pointer.

If collection occurs in order to satisfy a request, scanning stops when the request can be satisfied. If collection occurs as maintenance (or if storage runs low), all blocks / superblocks are scanned.

For any block found tagged *pending free*:

- the block is tagged *non-pending*,
- if collection is done as part of an allocation request, proceed with allocation from this block; otherwise
- the block inserted in the free block list for that superblock, or merged with the wilderness block if adjacent with it.

Whenever the scan stops (either after the request is honored, a predefined number of blocks has been scanned, or the scan wrapped around in the superblock), the "hint" pointer is updated to point to the next block. This allows future garbage collections to resume from that point.

After collection completes on a superblock, the free list and wilderness block are tested to see if the superblock has become empty. If the superblock is now empty:

- the superblock is removed from the bin,
- the superblock is added to the available list in the control block,
- the busy superblock count in the control block is decreased.

After all superblocks have been handled, and if an unmapping operation for slots is available in the implementation, then the following happens. The busy count in the control block is tested. If it becomes lower than some threshold (e.g. half the count of allocated superblocks), then some available superblocks are released (unmapped) to the environment.

Costs

Deallocation is constant time, worst case scenario is when the superblock is marked available and the storage released to the environment.

Allocation is constant time if the bin contains a non-full superblock. In this case it costs three memory loads and two stores.

If the bin is full, garbage collection occurs. If the bin is empty or garbage collection does not free superblocks, the cost is further increased by external heap allocation.

However the allocator is *scalable* in that allocations by different threads can run fully concurrently.

Hypotheses

We assume the following:

- tasks/threads are run within non-interruptible thread contexts that are recycled between tasks/threads (e.g. OpenMP tasks running on top of POSIX threads, or SVP threads running on top of Microgrid thread contexts)
- the environment supports virtual mapping of storage to arbitrary address ranges (at some granularity of ranges); and
- in any non-interruptible thread context, the thread code can query its own thread context identifier from the execution environment at a low cost; and
- the total (maximum) number of thread contexts is known.

From these assumptions, we can partition (a possibly sparse part of) the shared address space among all thread contexts.

If the partitioned address space is contiguous, aligned on a multiple of its size, and the set of all hardware thread identifiers is evenly spaced, the partitioning is trivial and the base address of each thread-specific region can be computed by simply shifting the bits of the thread identifier. Otherwise, a mapping table can be constructed at system startup.

From this point, we know we can construct the following two functions, available to program code and implemented either in hardware or in software, and computable at constant and minimal cost:

- `tls_base()` which, when run in a thread context, returns the base address of the thread-specific region of the address space;
- `tls_size()` which indicates the size of the thread-specific region.

> **Note**
>
> in the Microgrid, `tls_base` and `tls_size` are available in the ISA.

Slot size and alignment

The slot size, noted *CSZ*, and the base addresses of slots can be arbitrary specified under the following constraints:

- ***CSZ* must be larger than or equal to the combined size of 5 machine** words;
- *CSZ* must be smaller or equal to the size of the smallest thread-specific region according to the partitioning;
- *CSZ* must be a multiple of the granularity of the virtual storage mapping offered by the implementation (e.g. a page-size on page-based VM systems);
- *CSZ* must be a power of two;
- the base addresses of slots must be a multiple of *CSZ*.

The last two constraints exist to ensure that the base address of a slot/superblock can be deduced *intrinsically* from any address within the slot by simply masking the lower order bits of the inner address.

Example on the MT-Alpha Microgrid

The MT-Alpha Microgrid has a 64-bit address space. The operating system, ROM, configuration space and memory-mapped I/O is allocated on the lower half of the address space, i.e. all addresses with the most-significant bit (MSB) set to 1 can be used for thread-specific partitioning.

Given 2^P cores and 2^T threads per core, we can split this higher address space in 2^(P+T) regions of at most 2^(64-1-P-T) bytes each.

With Microgrid memory protection enabled ([svp7]), some address bits are reserved to identify the protection domain. The number of reserved bits is then P + F, if there are 2^F family entries per core, and the size of each thread-specific region becomes 2^(64-1-P-T-P-F).

Then in order to provide a C call stack to each thread, we further split each thread-specific region between an upper half (the stack) and a bottom half (the thread-specific heap).

The size remaining for heap allocation is then 2^(64-1-P-T-P-F-1).

With 256 cores, 256 threads per core, 32 families per core, this means `tls_size` = 8Gibytes.

With a "complete" Microgrid of 1024 cores, `tls_size` = 512Mibytes.

(As a comparison, on an "embedded Microgrid" based on the 32-bit SPARC ISA, with 32 cores, 32 threads per core, 8 families per core and no protection, `tls_size` = 1Mibytes.)

The Microgrid hardware maps storage with a cache line-sized granularity, i.e. 64 bytes.

We have thus:

- *CSZ* >= 5 * 8 = 40 (5 machine words)
- *CSZ* <= 2^(64-1-P-T-P-F-1) = 8Gibytes for the 256-core microgrid
- *CSZ* multiple of 64 (cache line, granularity of mapping)

Example on page-based Unix

Unix allows for a fresh private address space to each process. Assuming an SVP implementation supported by threads within a single Unix process, and assuming we can determine a contiguous area of the address space, the same technique applies as for the Microgrid.

For example, using currently-available 64-bit hardware and an open-source Unix implementation, the arbitrary VM mapping granularity is 4Kbytes (a page) and a test shows that a single process has access to a region of at least 2^40 bytes of unmapped address space before it starts allocating.

We have thus:

- *CSZ* >= 5 * 8 = 40 (5 machine words)
- *CSZ* <= (2^40 / P) = 64Gibytes if P = 16 (16 hardware threads)
- *CSZ* multiple of 4096 (1 page, granularity of mapping)

Using Sun Solaris 10 on the Niagara T2, we see that a process also has access to at least an area of 2^40 bytes of contiguous address space:

- *CSZ* >= 5 * 8 = 40 (5 machine words)
- *CSZ* <= (2^40 / P) = 16Gibytes (8 cores, 8 threads per core, therefore P = 64)
- *CSZ* multiple of 8192 (1 page, granularity of mapping on Solaris)

Slots and mappings

As supposed in the initial hypotheses, the underlying implementation supports mappings of storage to arbitrary address ranges.

We can support either implicit mappings (slots are mapped automatically on first access) or explicit mappings via a system service that allocates from a physical storage pool and maps the specified part of the (virtual) address space to the allocated storage.

Note

on the Microgrid, the mapping is implicit and performed on first access. On Unix, it can be performed by the system call `mmap(Addr, CSZ, PROT_READ|PROT_WRIT, MAP_ANON|MAP_FIXED, -1, 0)`.

We do not make any assumption on the availability of an unmapping operation (that releases storage), although it will can be used if it exists.

We also assume that the first slot can be mapped for every hardware thread. This can be enforced in various ways, e.g. by pre-mapping the first slot for every hardware thread when the SVP system starts up.

G Simple Example SL Program Utilising I/O API

```
#include <svp_hldrv.h>
#include <svp_lowdrvs.h>
#include <svp/iomacros.h>
#define putc putchar

char_t abuffer[100];
unsigned read_size=5;
device_node_t * tester;

//a simple callback thread
sl_def(acallback, void){
    sl_index(callbackid);
    puts("IO_TEST: Callback! \n"); puts("Callback ID: "); putu(callbackid);
    puts("\nCallback Values: ");
    static unsigned lastpos;
    int i; char t_char;
    for(i=lastpos; i< lastpos+read_size; i++){
      putc((char)abuffer[i]); putc(',');
    }
   lastpos+=read_size;
   putc('\n');
}
sl_enddef

//main io_test method
sl_def(t_main, void)
{
   sl_place_t io_place = IO_HLDRV_PLACE;
   sl_place_t io_ex_place = IO_HL_EX_PLACE;
  GET_EX_PID(io_ex_place);
  char_t readachar;

/*--------------------------------
          Device Initialisation
  -------------------------------- */
  puts("IO_TEST: Initialising Device Object...\n");
  INIT_DEV(1,io_place,io_ex_place,BLOCK_D,(ldrv_pntr)burstchar_drv,0,5,tester);
/* ------------------------------
          Main I/O Test
   ------------------------------ */
  puts("IO_TEST: Testing synchronous reads...\n");
  char_t tempbuff[read_size];
  READ_DEV(tester,tempbuff,read_size);
  puts("IO_TEST: Sync Read:");
  int s;
  for(s=0;s<read_size;s++) {putc((char)tempbuff[s]);putc(',');}
  putc('\n');

  puts("IO_TEST: Testing asynchronous reads...\n");
  READ_DEV_ASYNC(tester,buffer,read_size,(async_callback)&acallback);

  puts("End of I/O Test reached.\n");
  //should wait here indefinitely for async stuff
  while(1){;}
}
sl_enddef
```

H SL Standard Library (TR)

SL standard library

Key: sl5
Date: 2010-06-14
Status: Draft
Author: Raphael "kena" Poss
Source: svn+ssh://mike@mac-chris.science.uva.nl/Library/SVN/CSA/doc/notes/sl5.txt
Version: sl5.txt 3849 2010-06-14 16:36:57Z kena

Introduction

Standard library services in SL are available in two flavours:

- SVP thread functions suitable for use with `sl_create`; these require explicit create/sync constructs to be used;
- wrapper preprocessor macros that expand to more complex SL code containing data initialization and create/sync blocks.

Wrapper macros are intended to simplify writing SL programs, while "fitting" semantically in the traditional syntax patterns of C. For this purpose, macros are written to be either:

- statement-like, i.e. usable whenever a statement can be used in the language grammar;
- expression-like, i.e. usable whenever an expression can be used in the language grammar.

The syntactic role of each library service is described alongside with the functionality in the following sections.

Compiler utilities

The following definitions are available in `svp/compiler.h`:

- statement-like macro `nop()`: as its name implies causes the program to step forward with no side effect.
- expression-like macros `likely(X)` and `unlikely(X)` that can be used to guide branch optimization. See the provided macro definition and the description of `__builtin_expect` in the section "Other built-in functions provided by GCC" in the GCC documentation.

Communication with the environment

Text and data output

The services defined in note **[mgsim5]**_ are available in `svp/testoutput.h`.

Data input

The services defined in notes **[sl3]**_ are available in `svp/slr.h` and `svp/fibre.h`.

Console output

At the time of this writing the Microgrid simulator does not yet provide full-scale I/O, and are not equipped with an operating system rich enough to provide the full C I/O library. As the smallest common feature denominator across SVP implementations, it restricts the SL language to not provide the C I/O primitives, even if the language is derived from C.

However, the simulator does provide a single unbuffered character output channel. This can be simulated on other SVP implementations using character output on a process' standard output, so it has been chosen as a fundation for a small-scale character output library available to SL programs and a restricted subset of C's standard I/O library.

The custom output library provides the following interface:

- single character output: `svp_io_putc`;
- C string output: `svp_io_puts`;
- arbitrary bytes: `svp_io_write`;
- single floating point number: `svp_io_putf`;
- single signed integer: `svp_io_putn`;
- single unsigned integer: `svp_io_putun`;
- roman numerals for small unsigned integers: `roman`;
- formatted output: `svp_io_printf`;

For each of these services the actual thread function identifier is declared in `svp/io.h`, starting with "`svp_io`"; this is used by explicitly creating a family of a single thread using the identifier as thread function.

See below for examples.

See also Input/output (7.19) for standard C interfaces.

Character, string and bytes output

The interfaces are as follows:

- single character output:

```
sl_decl(svp_io_putc, void,
        sl_glparm(char, c));
```

- C string output:

```
sl_decl(svp_io_puts, void,
        sl_glparm(const char *, str));
```

- arbitrary bytes output:

```
sl_decl(svp_io_write, void,
        sl_glparm(void *, ptr),
        sl_glparm(size_t, size));
```

Floating point output

The following service is provided:

```
sl_decl(svp_io_putf, void,
        sl_glfparm(double, number),
        sl_glparm(unsigned, precision),
        sl_glparm(unsigned, base));

/* pseudo-declaration for the statement-like macro */
#include <svp/iomacros.h>
void putf(double number, unsigned precision);
```

When this service is used it prints the floating point number to the console output using the specified precision (number of digits) and base. When the wrapper macro `putf` is used, the base is set to ten.

The floating-point number is normalized before printing so that the output mantissa stays between zero and the specified (excluded). The sign is always printed.

Examples:

```
putf(10.4, 5); // outputs +1.0400E+1
putf(-1, 2);   // outputs -1.0E+1
putf(300, 1);  // outputs +3.E+2
```

The behavior of `svp_io_putf` becomes undefined when the base is set to 0 or 1.

The special values NaN, positive infinity and negative infinity are printed as `nan`, `+inf` and `-inf`, respectively.

Integer output

The following services are provided:

```
sl_decl(svp_io_putn, void,
        sl_glparm(int64_t, number),
        sl_glparm(unsigned, base));

sl_decl(svp_io_putun, void,
        sl_glparm(uint64_t, number),
        sl_glparm(unsigned, base));

/* pseudo-declarations for the statement-like macros */
#include <svp/iomacros.h>
void putn(int64_t number);
void putu(uint64_t number);
```

When this service is used it prints the intger number to the console output using the specified base. When the wrapper macros `putn` and `putu` are used, the base is set to ten.

Examples:

```
putu(42);    // outputs 42
putu('A');   // outputs 65
putn(-123);  // outputs -123
```

The behavior is undefined if the base is set to 0 or 1.

Roman numerals

As the Alpha-based SVP implementation does not provide hardware support for integer divide, the development of a software substitute (described above) required some primitive output for testing, to ensure the validity of the divide results. The resulting primitive is a thread function which outputs its parameter as roman numerals. The algorithm to output roman numerals does not require integer division.

This service is a special SL "extension" not defined in the C language, intended mostly for testing purposes. The thread function is declared in header `svp/roman.h` as follows:

```
sl_decl(roman, void, sl_glparm(short));
```

The roman numeral output does not use the substractive principle, i.e. the decimal value 9 is printed as `VIIII` instead of `IX`. A leading "-" is printed if the value is negative.

Data input

Interactive data input from the runtime environment is not supported as of this writing.

Initial data input (e.g. run-time program parameters) is supported via the external "slr" data input mechanism described in note [sl3]_.

Delegation and places

The following definitions can be reached by including `svp/delegate.h`:

- the declared type `sl_place_t`: opaque type suitable to declare any object (including synchronizing objects) that can hold a SVP place identifier. The only operations defined for this type are comparison (equality) and assignment.
- the constant `SVP_EXIT_NORMAL`: equivalent to `EXIT_NORMAL` in TC. Termination status when a family terminates normally.
- `SVP_EXIT_BREAK`: equivalent to `EXIT_BREAK` in TC. Termination status when a family terminates asynchronously due to the break operation.
- `SVP_EXIT_KILL`: equivalent to `EXIT_KILL` in TC. Termination status when a family terminates asynchronously due to the kill operation.
- `PLACE_LOCAL`: specifier for the local place.
- `PLACE_DEFAULT`: specifier for the default place. (note, this is the default when no place is specified in `sl_create`)
- `PLACE_GROUP`: specifier for the enclosing place (after `PLACE_LOCAL` restricts a current place).

Additionally, the services defined in note [sl7]_ are available in `svp/sep.h`.

Compatibility with the C99 library

Support for features from the C99 library is *unspecified* unless stated explicitly in the following sections.

Diagnostics (7.2)

The standard header `assert.h` and its services are available in SL.

Character handling (7.4)

The standard header `ctype.h` and its services are available in SL.

Errors (7.5)

The standard header `errno.h` and its services are available in SL.

> **Note**
>
> The value `errno` declared therein is shared by all threads and may not be kept consistent between uses when modified by concurrent threads.

Characteristics of floating-point types (7.7)

The standard header `float.h` and its services are available in SL.

Sizes of integer types (7.10)

The standard header `limits.h` and its services are available in SL.

Mathematics (7.12)

The standard header `math.h` is available to SL programs; it supports the C standard services except for the following:

- any function operating with the `long double` data type;
- the `FP_CONTRACT` pragma.

> **Note**
>
> Programs using the C math library should be linked with `-lm`.

> **Warning**
>
> Due to an incomplete underlying implementation, the `tgamma` function may return incorrect results.

Variable arguments (7.15)

The standard header `stdarg.h` and its services are available in SL.

Boolean types and values (7.16)

The standard header `stdbool.h` and its services are available in SL.

Common definitions (7.17)

The standard header `stddef.h` and its services are available in SL.

Integer types (7.18)

The standard header `stdint.h` and its services are available in SL.

Input/output (7.19)

The standard header `stdio.h` is available to SL programs, but support is restricted to the following:

- the declared types `size_t`, `FILE`, `fpos_t`,
- the macros `NULL`, `EOF`, `BUFSIZ`, `FILENAME_MAX`,
- the expressions `stderr` and `stdout`,
- the byte output functions `fputc`, `fputs`, `fwrite`, `putc`, `putchar`, `puts`, `fprintf`, `printf`, `vfprintf`, `vprintf`,
- the string formatting functions `snprintf`, `sprintf`, `vsnprintf`, `vsprintf`,
- the error formatting function `perror`.

> **Note**
>
> `stdout` is unbufferred.

> **Note**
>
> See also Supported C extensions below.

General utilities (7.20)

The standard header `stdlib.h` is available to SL programs, but support is restricted to the following:

- the declared type `size_t`,
- the macros `NULL`, `EXIT_FAILURE` and `EXIT_SUCCESS`;
- integer-string conversion functions (`atoi`, `atol`, `atoll` from 7.20.1.2, `strtol`, `strtoll`, `strtoul`, `strtoull` from 7.20.1.4);
- memory management functions (`malloc`, `realloc`, `free`, `calloc` from 7.20.3)
- the `abort` function (7.20.4.1)
- the `getenv` function (7.20.4.5)
- the `exit` and `_Exit` functions (7.20.4.3, 7.20.4.4)

> **Note**
> See also Supported C extensions below.

String handling (7.21)

The standard header `string.h` is available to SL programs, but support is restricted to the following:

- copying functions (`memcpy`, `memmove`, `strcpy`, `strncpy` from 7.21.2)
- the concatenation functions `strcat`, `strncat` (7.21.3)
- the comparison function `strncmp` (7.21.4)
- the search function `strchr` (7.21.5)
- the misc functions `memset`, `strerror` and `strlen` (7.21.6)

> **Note**
> See also Supported C extensions below.

Time (7.23)

The standard header `time.h` is available to SL programs, but support is restricted to the following:

- `CLOCKS_PER_SEC`;
- the declaration of the `clock()` function.

Supported C extensions

Allocation from the stack frame

The non-standard (but nevertheless common) pseudo-function `alloca` is available in `alloca.h`:

```
void* alloca(size_t);
```

This function allocates space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the function from which `alloca` is called returns, or when the thread where it is called terminates.

Note

It is not valid to pass a pointer returned by `alloca` to C's `free` function.

Note

The behavior of `alloca` in combination with C99's variable length arrays (VLAs) is left *unspecified* in this document.

Note

As this function is a compiler intrinsic in most implementations, the name `alloca` *may* be a macro and may not have an address.

Extra string handling

The following popular BSD and POSIX extensions are available from `string.h` and `strings.h`:

- safe string manipulation: `stpcpy`, `stpncpy`, `strnlen`, `strlcpy`, `strlcat`;
- MT-safe `strerror_r`;
- `strdup`;
- `bcopy` (old BSD interface to `memmove`), `bzero` (old BSD interface to `memset`).