RC64: High Performance Rad-Hard Manycore

Ran Ginosar, Peleg Aviely, Fredy Lange and Tsvika Israeli

Ramon Chips, Ltd., 5 HaCarmel Street, Yoqneam Illit 2069201, Israel

[ran, peleg, fredy, tsvika]@ramon-chips.com

Abstract

RC64 is a rad-hard manycore DSP combining 64 VLIW/SIMD DSP cores, lock-free shared memory, a hardware scheduler and a task-based programming model. The hardware scheduler enables fast scheduling and allocation of fine grain tasks to all cores.

I. INTRODUCTION

Multiple core architectures are divided into multi-cores and many-cores. Multi-cores, ranging from rad-hard Gaisler/ Ramon Chips' LEON3FT dual-core GR712RC to commercial ARM Cortex A9 and Intel Xeon, typically provide some form of cache coherency and are designed to execute many unrelated processes, governed by an operating system such as Linux. In contrast, many-cores such as Tilera TilePro, Adapteva's Epiphany, NVidia GPU, Intel Xeon Phi and Ramon Chips' RC64, execute parallel programs specifically designed for them and avoid operating systems, in order to achieve higher performance and higher power-efficiency.

Many-core architectures come in different flavors: a two-dimensional array of cores arranged around a mesh NoC (Tilera and Adapteva), GPUs and other manycores with clusters of cores (Kalray), and rings. This paper discusses the Plural architecture [12]—[16] of RC64 [17], in which many cores are interconnected to a many-port shared memory rather than to each other (Figure 1).

Many cores also differ on their programming models, ranging from PRAM-like shared memory through CSP-like message-passing to dataflow. Memory access and message passing also relate to data dependencies and synchronization—locks, bulk-synchronous patterns and rendezvous. RC64 architecture employs a strict shared memory programming model.

The last defining issue relates to task scheduling—allocating tasks to cores and handling task dependencies. Scheduling methods include static (compile time) scheduling, dynamic software scheduling, architecture-specific scheduling (e.g., for NoC), and hardware schedulers, as in RC64, in which data dependencies are replaced by task dependencies in order to enhance performance and efficiency and to simplify programming.

As a processor designed for operation in harsh space environment, RC64 is based on rad-hard technology and includes several mechanisms to enhance its fault tolerance, such as EDAC, and to handle fault detection, isolation and recovery (FDIR).

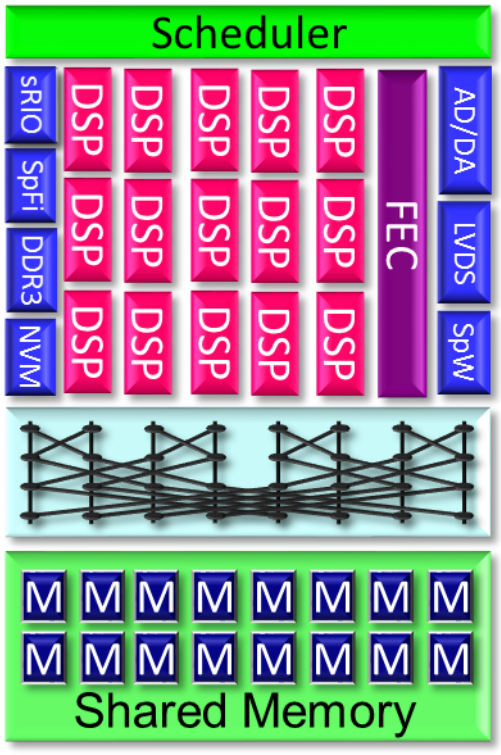

Figure 1. RC64 Many-Core Architecture. 64 DSP cores, modem accelerators and multiple DMA controllers of I/O interfaces access the multibank shared memory through a logarithmic network. The hardware scheduler dispatches fine grain tasks to cores, accelerators and I/O.

II. Related Work

GR712RC, an early dual-core rad-hard space processor was introduced by Ramon Chips and Cobham Gaisler [1][2]. Other multi-core architectures, not intended for space, include ARM Cortex A9 [3] and Intel Xeon. Many core architectures include the mesh-tiled Tilera [4][5] and Adapteva [6], NVidia GPU [7], Intel ring-topology Xeon Phi [8] and dataflow clusters by Kalray [9]. The research XMT manycore [10] is PRAM-inspired and employs hardware scheduling, similar to RC64. It employs declarative parallelism to direct scheduling [11]. The Plural architecture and its RC64 incarnation are discussed in [12]—[17] and is the subject of the MacSpace European FP7 research project [18]. An early hardware scheduler is reported in [19]. The baseline multistage interconnection network has been introduced in [20]. Example of SDR modem implementation on RC64 and simulated performance results are given in [26].

Other efforts to introduce rad-hard manycores for space include the FPGA-based AppSTAR at Harris [22], Maestro at Boeing [23] and RADSPEED at BAE Systems [24].

III. RC64 Architecture

This section presents the Plural architecture of RC64 (Figure 1). RC64 architecture defines a shared-memory single-chip many-core. The many-core consists of a hardware synchronization and scheduling unit, 64 DSP cores, and a shared on-chip memory accessible through a high-performance logarithmic interconnection network. The cores contain instruction and data caches, as well as a private 'scratchpad' memory. The data cache is flushed and invalidated by the end of each task execution, guaranteeing consistency of the shared memory. The cores are designed for low power operation using 'slow clock' (typically slower than 500 MHz). Performance is achieved by high level of parallelism rather than by sheer speed, and access to the on-chip shared memory across the chip takes only a small number of cycles.

The on-chip shared memory is organized in a large number of banks, to enable many ports that can be accessed in parallel by the many cores, via the network. To reduce collisions, addresses are interleaved over the banks. The cores are connected to the memory banks by a multi-stage many-to-many interconnection network. The network detects access conflicts contending on the same memory bank, proceeds serving one of the requests and notifies the other cores to retry their access. The cores immediately retry a failed access. Two or more concurrent read requests from the same address are served by a single read operation and a multicast of the same value to all requesting cores. As explained in the next section, there is no need for any cache coherency mechanism.

The CEVA X1643 DSP core comprises the following parts. The computation unit consists of four multiplier-accumulators (MAC) of 16-bit fixed point data, supporting other precisions as well, and a register file. Ramon Chips has added a floating point MAC. The data addressing units includes two load-store modules and address calculation. The data memory unit consists of the data cache, AXI bus interface, write buffers for queuing write-through transactions and a scratchpad private memory. The program memory unit is the instruction cache. Other units support emulation and debug and mange power gating. Thus, the DSP core contains three memories: an instruction cache, a write-through data cache and a scratchpad private memory.

Implemented in 65nm CMOS and designed for operation at 300 MHz, RC64 is planned to achieve 38 GFLOPS (single precision) and 76 GMAC (16-bit). With 12 high speed serial links operating at up to 5 Gbps in each direction, a total bandwidth of 120 Gbps is provided. Additional high bandwidth is enabled for memories (25 Gbps DDR3 interface of 32 bit at 800 Mword/s with additional 16 bits for ECC) and for high performance ADC and DAC (38 Gbps over 48 LVDS channels of 800 Mbps). The device is planned to dissipate less than 10 Watt in either CCGA or PBGA 624 column or ball grid array packages.

IV. RC64 Programming Model

The Plural PRAM-like programming model of RC64 is based on non-preemptive execution of multiple sequential tasks. The programmer defines the tasks, as well as their dependencies and priorities which are specified by a (directed) *task graph*. Tasks are executed by cores and the task graph is 'executed' by the scheduler.

In the Plural shared-memory programming model, concurrent tasks cannot communicate. A group of tasks that are allowed to execute in parallel may share read-only data but they cannot share data that is written by any one of them. If one task must write into a shared data variable and another task must read that data, then they are *dependent*—the writing task must complete before the reading task may commence. That dependency is specified as a directed edge in the task graph, and enforced by the hardware scheduler. Tasks that do not write-share data are defined as *independent*, and may execute concurrently. Concurrent execution does not necessarily happens at the same time—concurrent tasks may execute together or at any order, as determined by the scheduler.

Some tasks, typically amenable to independent data parallelism, may be *duplicable*, accompanied by a *quota* that determines the number of instances that should be executed (declared parallelism [11]). All instances of the same duplicable task are mutually independent (they do not write-share any data) and concurrent, and hence they may be executed in parallel or in any arbitrary order. These instances are distinguishable from each other merely by their *instance number*. Ideally, their execution time is short (fine granularity). Concurrent instances can be scheduled for

execution at any (arbitrary) order, and no priority is associated with instances.

Each task progresses through at most four states (Figure 2). Tasks without predecessors (enabled at the beginning of program execution) start in the *ready* state. Tasks that depend on predecessor tasks start in the *pending* state. Once all predecessors to a task have completed, the task becomes *ready* and the scheduler may schedule its instances for execution and allocate (dispatch) the instances to cores. Once all instances of a task have been allocated, the task is *All allocated.* And once all its instances have terminated, the task moves into the *terminated* state (possibly enabling successor tasks to become *ready*).

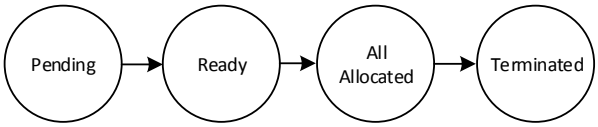

Figure 2. Task State Graph

Many-flow pipelining facilitates enhanced core utilization in streamed signal processing. Consider the task graph examples for executing JPEG2000 image compression and the processor utilization charts of Figure 3. In (a), five tasks A-E are scheduled in sequence. Tasks B and D are duplicable with a large number of instances, enabling efficient utilization of 64 cores. Tasks A,C,E, on the other hand, are sequential. Execution time of compressing one image is 160 time units, and overall utilization, reflected by the ratio of colored area to the 64×160 rectangle, is 65%. The core utilization chart (on the right) indicates the number of busy cores over time, and different colors represent different tasks. In the many-flow task graph (Figure 3b), a pipeline of seven images is processed. During one iteration, say iteration *k*, the output stage sends compressed image *k*, task E processes image *k*+1, task D computes the data of image *k*+2, and so on. Notice that the sequential tasks A,C,E are allocated first in each iteration, and duplicable instances occupy the remaining cores. A single iteration takes 95 time units and the latency of a single image is extended to five iterations, but the throughput is enhanced and the core utilization chart now demonstrates 99% core utilization.

Data dependencies are expressed (by the programmer) as task dependencies. For instance, if a variable is written by task t_w and must later be read, then reading must occur in a group of tasks $\{t_r\}$ and $t_w \rightarrow \{t_r\}$. The synchronization action of completion of t_w prior to any execution of tasks $\{t_r\}$ provides the needed barrier.

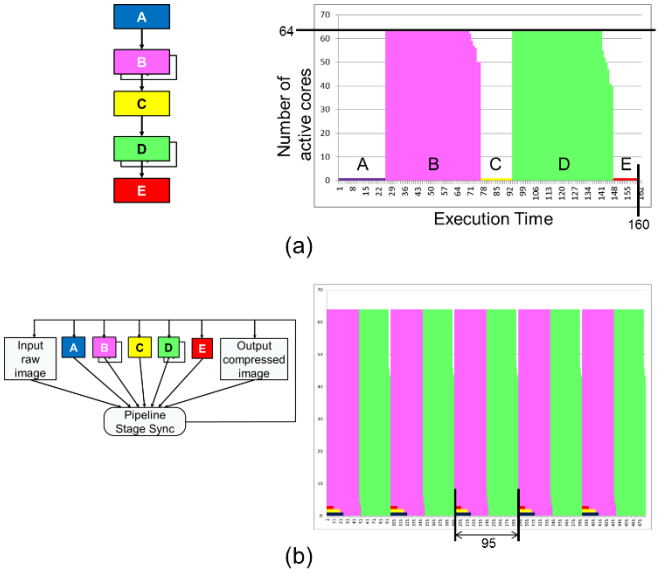

Figure 3. Many-flow pipelining: (a) task graph and single execution of an image compression program, (b) many-flow task graph and its pipelined execution

V. RC64 HARDWARE SCHEDULER

The hardware scheduler assigns tasks to cores for execution. The scheduler maintains two data structures, one for managing cores (Figure 4) and the other for managing tasks (Figure 5). Core and task state graphs are shown in Figure 6 and Figure 2, respectively.

The hardware scheduler operates as follows. At start, all cores are listed as Idle and the task graph is loaded into the first three columns of the Task Management Table. The scheduler loops forever over its computation cycle. On each cycle, the scheduler performs two activities: allocating tasks for execution, and handling task completions.

Core #	State	Task #	Instance #	…	…
0					
1					
2					
…					

Figure 4. Core Management Table

Task #	Duplication quota	Dependencies	State	# allocated instances	# terminated instances
0					
1					
2					
…					

data from task graph

Figure 5. Task Management Table

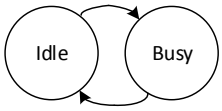

Figure 6. Core State Graph

To allocate tasks, the scheduler first selects ready tasks from the Task Management Table. It allocates each such task to idle cores by changing the task state to *All Allocated* (if the task is regular, or if all duplicable instances have been dispatched), by increasing the count of allocated instances in the Task Management Table, and by noting the task number (and instance number, for duplicable tasks) in the Core Management Table. Finally, task/instance activation messages are dispatched to the relevant cores. The activation message for a specific core includes the code entry address and (in case of a duplicable instance) the instance ID number.

To handle task completions, the scheduler collects termination messages from cores that have completed task executions. It changes the state of those cores to *Idle*. For regular tasks, the task state is changed to *Terminated*. For duplicable tasks, the counter of terminated tasks in the Task Management Table is incremented, and if it has reached the quota value then the state of that task is changed to *Terminated*. Next, the scheduler updates the Dependencies entry of each task in the table which depends on the terminated task: the arrival of that token is noted, the dependency condition is recomputed, and if all precedencies of any task have been fulfilled then the state of that task is changed to *Ready*, enabling allocation and dispatch in subsequent scheduler computation cycles.

The *scheduler capacity*, namely the number of simultaneous tasks which the scheduler is able to allocate or terminate during each computation cycle, is limited. Any additional task allocations and task termination messages beyond scheduler capacity wait for subsequent cycles in order to be processed. A core remains idle from the time it issues a termination message until the next task allocation arrives. That idle time comprises not only the delay at the scheduler (wait and processing times) but also any transmission latency of the termination and allocation messages over the scheduler-to-cores network.

The allocation and termination algorithms are shown in Figure 7.

Scheduling efficiency depends on the ratio of scheduling latency (reflected in idle time of cores) to task execution time. Extremely fine grain tasks (e.g., those executing for 1~100 cycles) call for very short scheduling latencies (down to zero cycles) to be efficient. Alternatively, speculative advanced scheduling may fill queues attached to each core so that the core can start executing a new instance once it has completed a previous instance (see [16] for such an analysis). However, typical tasks tend to incur compiled overhead (prologue and epilogue code sequences generated by even the most efficient optimizing compilers), and typical programming practices of parallel tasks tend to avoid the shortest tasks, resulting in average task duration exceeding 100 cycles. With average scheduling latency of only 10-20 cycles, enabled by hardware implementation, we obtain execution efficiency close to 99%.

The hardware scheduler is implemented as custom logic in RC64. Two other possibilities will be considered in future generations, one based on two content-addressable memory (CAM) arrays implementing the two management tables, and another implementation as software executing on a dedicated fast core with its dedicated high throughput memory.

ALLOCATION

1. Choose a *Ready* task (according to priority, if specified)
2. While there is still enough scheduler capacity and there are still *Idle* cores
    a. Identify an Idle core
    b. Allocate an instance to that core
    c. Increase counter of allocated task instances
    d. If # allocated instances == quota, change task state to *All Allocated* and continue to next task (step 1)
    e. Else, continue to next instance of same task (step 2)

TERMINATION

1. Choose a core which has sent a termination message
2. While there is still enough scheduler capacity
    a. Change core state to *Idle*
    b. Increment # terminated instances
    c. If # terminated instances == quota, change task state to *Terminated*
    d. Recompute dependencies for all other tasks that depend on the terminated task, and where relevant change their state to *Ready*

Figure 7. Allocation (top) and termination (bottom) algorithms

A special section of the scheduler schedules High Priority Tasks (HPTs), which are designed as 'interrupt handling routines' to handle hardware interrupts. As explained in Section VII, all I/O interfaces (including interfaces to accelerators) are based on DMA controllers that issue interrupts once completing their action. The most urgent portion of handling the interrupt is packaged as a HPT, and less urgent parts are formulated as a normal task. HPT is dispatched immediately and pre-emptively by the scheduler. Each core may execute one HPT, and one HPT does not pre-empt another HPT. Thus, a maximum of 64 HPTs may execute simultaneously. RC64 defines fewer than 64 different HPTs, and thus there is no shortage of processors for prompt invocation of HPTs.

VI. RC64 Networks on Chip

RC64 contains two specialized Networks on Chip (NOCs), one connecting the scheduler to all cores and other schedulable entities (DMA controllers and accelerators), and a second NOC connecting all cores and other data sources (DMA controllers) to the shared memory.

A. Scheduler NOC

The scheduler-to-cores NOC employs a tree topology. That NOC off-loads two distributed functions from the scheduler, task allocation and task termination.

The distributed task allocation function receives clustered task allocation messages from the scheduler. In particular, a task allocation message related to a duplicable task specifies the task entry address and a range of instance numbers that should be dispatched. The NOC partitions such a clustered message into new messages specifying the same task entry address and sub-range of instance numbers, so that the sub-ranges of any two new messages are mutually exclusive and the union of all new messages covers the same range of instance numbers as the original message. The NOC nodes maintain Core and Task Management Tables which are subsets of those tables in the scheduler (Figure 4 and Figure 5, respectively), to enable making these distributed decisions.

The distributed task termination process complements task allocations. Upon receiving instance terminations from cores or subordinate nodes, a NOC node combine the messages and forwards a more succinct message specifying ranges of completed tasks.

B. Shared Memory NOC

The larger NOC of RC64 connects 64 cores, tens of DMA controllers and hardware accelerators to 256 banks of the shared memory. To simplify layout, floor-planning and routing, we employ a Baseline logarithmic-depth multistage interconnection network [20], symbolically drawn in Figure 1. Some of the NOC switch stages are combinational, while others employ registers and operate in a pipeline. Two separate networks are used, one for reading and another one for writing. The read networks accesses and transfers 16 bytes (128 bits) in parallel, matching cache line size and serving cache fetch in a single operation. The write network is limited to 32 bits, compatible with the write-through mechanism employed in the DSP cores. Writing smaller formats (16 and 8 bits) is also allowed.

VII. RC64 Accelerators and I/O

Certain operations cannot be performed efficiently on programmable cores. Typical examples require bit level manipulations that are not provided for by the instruction set, such as used for error correction (LDPC, Turbo code, BCH, etc.) and for encryption. RC64 offers two solutions. First, several accelerators for pre-determined computations (such as LDPC and Turbo Coding, useful in DVB-S2 and DVB-RCS for space telecommunications) are included on chip. They are accessible only through shared memory, as follows. First, the data to be processed by the accelerator are deposited in shared memory. Next, the accelerator is invoked. Data is fetched to the accelerator by a dedicated DMA controller, and the outcome is sent back to shared memory by a complementing second DMA controller. This mode of operation decouples the accelerator from the cores and eliminates busy waiting of cores.

The second possibility is to employ an external acceleration on either an FPGA or an ASIC. High speed serial links on RC64 enable efficient utilization of such external acceleration. This mode offers scalability and extendibility to RC64.

All input / output interfaces operate asynchronously to the cores. Each interface is managed by one DMA controller for input and a second DMA controller for output. Many different types of I/O interfaces are available in RC64, including slow GPIO and SpaceWire links, high rate DDR2/DDR3 and ONFI flash EDAC memory interfaces (error detection and correction is carried out at the I/O interfaces, offloading that compute load from the cores), high speed serial links (implementing SpaceFibre [25], serial Rapid IO and proprietary protocols) and 48-link LVDS port useful for ADCs, DACs and other custom interfaces.

All DMA controllers are scheduled by the scheduler, submit interrupt signals to the scheduler (as explained in Section V above), and read and write data directly to the shared memory through the NOC (see Section VI above). The system software required for managing I/O is described in Section VIII below.

VIII. RC64 System Software

The system run-time software stack is shown schematically in Figure 8. The boot sequence library is based on the boot code of the DSP core. It is modified to enable execution by many cores in parallel. Only one of the cores performs the shared memory content initialization. The boot code

includes DSP core self-test, cache clearing, memory protection configuration and execution status notification to an external controlling host.

The Runtime Kernel (RTK) performs the scheduling function for the DSP core. It interacts with the hardware scheduler, receives task allocation details, launches the task code and responds with task termination when the task is finished. The RTK also initiates the power down sequence when no task is received for execution.

The first task allocated by the scheduler is responsible for loading the application task graph into the scheduler. This code is automatically generated during a pre-compile stage according to the task graph definition. Application tasks are allocated after the initialization task is finished.

Certain library routines manage EDAC for memories, encapsulate messaging and routing services to off-chip networking (especially over high speed serial SpaceFibre links), respond to commands received from an external host (or one of the on-chip cores, playing the role of a host), perform FDIR functions, and offer some level of virtualization when multiple RC64 chips are employed in concert to execute coordinated missions.

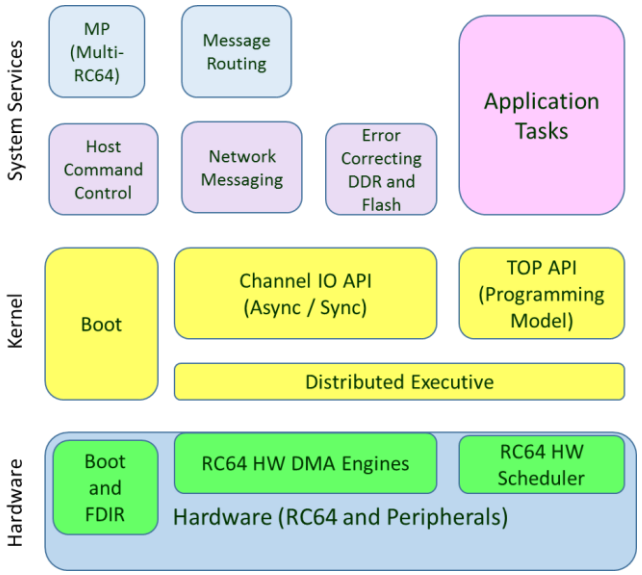

Figure 8. RC64 Run Time Software. The kernel enables boot, initialization, task processing and I/O. Other services include execution of host commands, networking and routing, error correction and management of applications distributed over multiple RC64 chips

Other components of the RTK manage I/O and accelerators. Configuring the interfaces requires special sequences such as link detection and activation, clock enabling, DMA configuration, etc. Each interface has its own set of parameters according to the required connectivity, storage type, data rate and so on.

Figure 9 demonstrate the hardware-kernel-application sequence of events in the case of an input of a predefined data unit over a stream input link. The DMA controller, previously scheduled, stores input data into a pre-allocated buffer in memory (step 1). Upon completion, it issues an interrupt (step 2). A HPT is invoked (step 3, see Section V) and stores pointers and status in shared memory, effectively enqueuing the new arrival (step 4). It ends up by issuing a 'software event' to the scheduler (step 5). Eventually, the scheduler dispatches a task that has been waiting for that event (step 6). That task can consume the data and then dequeue it, releasing the storage where the data was stored (step 7). Other I/O operations are conducted similarly.

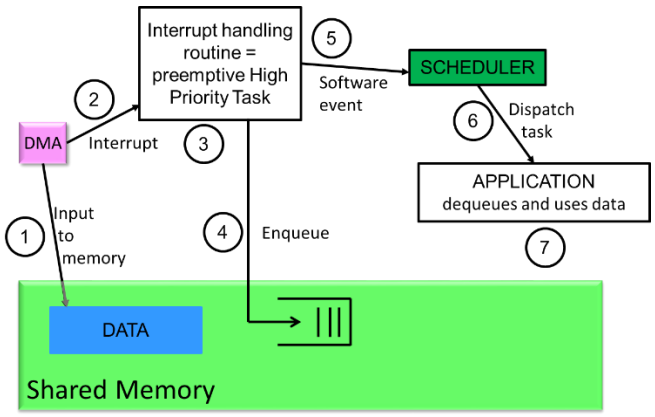

Figure 9. Event sequence performing stream input

IX. RC64 Software Development Tools

RC64 SDK enables software development, debug and tuning, as shown in Figure 10. The IDE tool chain includes a C/C++ compiler for the DSP core, an assembler, a linker, and a library of DSP functions customized for the core, taking full advantage of its VLIW capability (computing and moving data at the same time) and SIMD (performing several multiply and accumulate operations in parallel).

RC64 Parallel programming is supported by the task compiler, which translates the task graph for the scheduler, a many-task emulator (MTE) that enables efficient development of parallel codes on personal computers, and a many-core debugger, which synchronizes debug operations of all cores. The RC64 parallel simulator is cycle accurate, fully simulating the cores as well as all other hardware components on the chip.

The profiler provides complete record of parallel execution on all 64 cores. The event recorder generates traces with time stamps of desired events. The kernel and libraries are described in Section VIII above.

X. RC64 Radiation Hardness and FDIR

RC64 will be implemented in 65nm CMOS using RadSafe™ rad-hard-by-design (RHBD) technology and library [21]. RadSafe™ is designed for a wide range of space missions, enabling TID tolerance to 300 kRad(Si), no latchup and very low SEU rate. All memories on chip are protected by various means and varying levels of error correction and detection. Special protection is designed for registers that hold data for extended time, such as configuration registers. The two external memory interfaces, to DDR2/DDR3 and to ONFI flash memories, implement several types of EDAC. For instance, ten flash memory chips can be connected for eight byte wide datapath and two flash devices for storing Reed Solomon ECC.

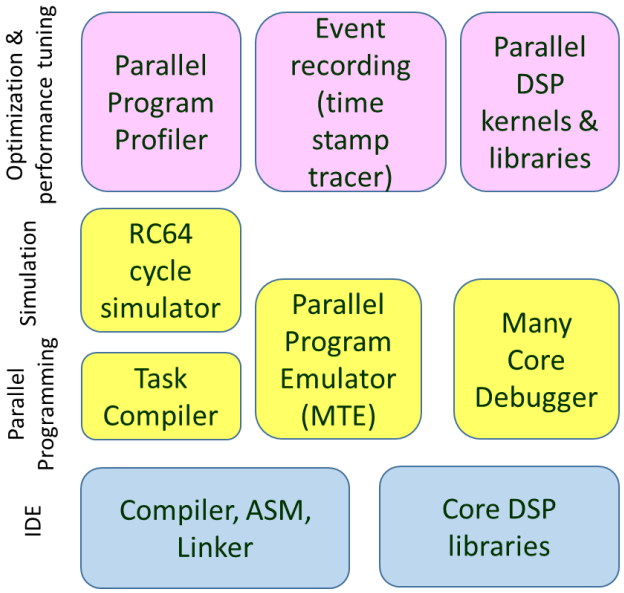

Figure 10. RC64 Software Development Kit.

RC64 implements extensive means for fault detection, isolation and recovery (FDIR). An external host can reset, boot and scrub the device through dual RMAP SpaceWire ports. RC64 contains numerous error counters and monitors that collect and report error statistics. Trace buffers, allocated in shared memory as desired, enable rollback and analysis (in addition to helping debug). Faulty sub-systems may be shut down and the scheduler is designed to operate with partial configurations.

XI. Conclusions

RC64 is a many core architecture suitable for use in space. It is designed for simplified PRAM-like shared memory programming and high performance at low power. RC64 goal is to enable future software-defined satellites in all space endeavors. RC64 is presently under design and all performance figures reported herein and in [26] are based on simulations. RC64 is planned for availability before the end of the decade. RC64 R&D project is funded by Israel Space Agency and by the European Union.

XII. Acknowledgements

The financial support of the Israel Space Agency, the Israel Ministry of Defense, the Israel Aerospace Industry and the European Union (Seventh Framework Programme grant agreement 607212) is greatly appreciated. Itai Avron has contributed to early versions of this paper.

XIII. References

[1] Sturesson, F., J. Gaisler, R. Ginosar, and T. Liran. "Radiation characterization of a dual core LEON3-FT processor." In Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on, pp. 938-944. IEEE, 2011.

[2] Habinc, S., K. Glembo, and J. Gaisler. "GR712RC-The Dual-Core LEON3FT System-on-Chip Avionics Solution." In DASIA 2010 Data Systems In Aerospace, vol. 682, p. 8. 2010.

[3] Jacquet, David, Frederic Hasbani, Philippe Flatresse, Richard Wilson, Franck Arnaud, Giorgio Cesana, Thierry Di Gilio et al. "A 3 GHz dual core processor ARM cortex TM-A9 in 28 nm UTBB FD-SOI CMOS with ultra-wide voltage range and energy efficiency optimization." Solid-State Circuits, IEEE Journal of 49, no. 4 (2014): 812-826.

[4] Villalpando, Carlos Y., Andrew E. Johnson, Raphael Some, Jacob Oberlin, and Steven Goldberg. "Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander." In Aerospace Conference, 2010 IEEE, pp. 1-9. IEEE, 2010.

[5] Wentzlaff, David, et al. "On-chip interconnection architecture of the tile processor." IEEE micro 5 (2007): 15-31.

[6] Varghese, Anitha, Ben Edwards, Gaurav Mitra, and Alistair P. Rendell. "Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor." In Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, pp. 984-992. IEEE, 2014.

[7] Nickolls, John, and William J. Dally. "The GPU computing era." IEEE micro 2 (2010): 56-69.

[8] Heinecke, Alexander, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, Grigorios Chrysos, and Pradeep Dubey. "Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor." In Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 126-137. IEEE, 2013.

[9] De Dinechin, Benoît Dupont, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. "Time-critical computing on a single-chip massively parallel processor." In Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, pp. 1-6. IEEE, 2014.

[10] Wen, Xingzhi, and Uzi Vishkin. "Fpga-based prototype of a pram-on-chip processor." In Proceedings of the 5th conference on Computing frontiers, pp. 55-66. ACM, 2008.

[11] Tzannes, Alexandros, George C. Caragea, Uzi Vishkin, and Rajeev Barua. "Lazy scheduling: A runtime adaptive scheduler for declarative parallelism." ACM Transactions on Programming Languages and Systems (TOPLAS) 36, no. 3 (2014): 10.

[12] Bayer, Nimrod, and Ran Ginosar. "High flow-rate synchronizer/scheduler apparatus and method for multiprocessors." U.S. Patent 5,202,987, issued April 13, 1993.
[13] Bayer, Nimrod, and Ran Ginosar. "Tightly Coupled Multiprocessing: The Super Processor Architecture." In Enabling Society with Information Technology, pp. 329-339. Springer Japan, 2002.
[14] Bayer, Nimrod, and Aviely Peleg. "Shared memory system for a tightly-coupled multiprocessor." U.S. Patent 8,099,561, issued January 17, 2012.
[15] Avron, Itai, and Ran Ginosar. "Performance of a hardware scheduler for many-core architecture." In 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 151-160. IEEE, 2012.
[16] Avron, Itai, and Ran Ginosar. "Hardware Scheduler Performance on the Plural Many-Core Architecture." In Proceedings of the 3rd International Workshop on Many-core Embedded Systems, pp. 48-51. ACM, 2015.
[17] Ran Ginosar and Peleg Aviely, RC64 – Many-Core Communication Processor for Space IP Router. In Proceedings of International Astronautical Conference, pp. IAC-15-B2.6.1, Jerusalem, Israel, Oct. 2015.
[18] http://www.macspace.eu/
[19] Crummey, T. P., D. I. Jones, P. J. Fleming, and W. P. Marnane. "A hardware scheduler for parallel processing in control applications." In Control, International Conference on, vol. 2, pp. 1098-1103. IET, 1994.
[20] Wu, Chuan-Lin, and Tse-Yun Feng. "On a class of multistage interconnection networks." Computers, IEEE Transactions on, vol. C-29, no. 8, pp. 694-702, 1980.
[21] Liran, Tuvia, Ran Ginosar, Fredy Lange, Peleg Aviely, Henri Meirov, Michael Goldberg, Zeev Meister, and Mickey Oliel. "65nm RadSafe™ technology for RC64 and advanced SOCs." (2015).
[22] Beadle, Edward R., and Tim Dyson. "Software-Based Reconfigurable Computing Platform (AppSTAR TM) for Multi-Mission Payloads in Spaceborne and Near-Space Vehicles." In International Conference on Reconfigurable Systems and Algorithms, ERSA 2012.
[23] Malone, Michael. "OPERA RHBD multi-core." In Military/Aerospace Programmable Logic Device Workshop (MAPLD 2009). 2009.
[24] Marshall, Joseph, Richard Berger, Michael Bear, Lisa Hollinden, Jeffrey Robertson, and Dale Rickard. "Applying a high performance tiled rad-hard digital signal processor to spaceborne applications." In Aerospace Conference, 2012 IEEE, pp. 1-10. IEEE, 2012.
[25] Parkes, Steve, Chris McClements, David McLaren, Albert Ferrer Florit, and Alberto Gonzalez Villafranca. "SpaceFibre: A multi-Gigabit/s interconnect for spacecraft onboard data handling." In Aerospace Conference, pp. 1-13. IEEE, 2015.
[26] Aviely, Peleg, Olga Radovsky and Ran Ginosar. "DVB-S2 Software Defined Radio Modem on the RC64 Manycore DSP." In DSP Day, 2016.