Volume 3, Issue 9, September 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Simulating Cluster Node Hardware Accelerators Using Sniper Simulator

Alexander Yu. Drozdov
Faculty of Radio Engineering and Cybernetics, MIPT alexander.y, Russia

Yuriy N. Fonin
Faculty of Radio Engineering and Cybernetics, MIPT, Russia

Denis A. Kondrushkin
Faculty of Radio Engineering and Cybernetics, MIPT, Russia

Abstract – The paper describes an approach to design and modeling of architectures with hardware accelerators. It presents an example of the Sniper-based hardware simulation model, which includes accelerator models in addition to x86-cores. The paper also introduces a technology of the accelerator driver development using Sniper simulator.

Keywords – architecture, microprocessor, modeling, accelerator, Sniper, simulation.

I. Introduction

Presently distributed cluster computations are increasingly used in resolution of various research and practical problems. However, clusters are quite often developed for resolution of some set of problems of a single or of as little as two or three considerably narrow subject areas. In such cases the attention is paid to the issue of development of clusters which would feature the best capacity for the particular set of problems under cost and input power constraints. A way to such an optimization is to install one or more hardware accelerators in parallel with the main processor. Such accelerators could be custom microprocessors, programmable logic microcell arrays that perform a particular set of functions. The accelerators of clusters based on x86-compatible processors are typically connected to the core processor via PCI or PCI-express protocols. At a first approximation it is possible to assume that a cluster node is a system comprising one or more core (x86) computational kernels and one or more hardware accelerators. Accelerators typically enable speeding-up target programs (i.e. programs the cluster is designed to execute) substantially.

In our instance the accelerator is a set of four processors functionally compatible with TigerSHARC TS 201. The processors are interconnected by means of so-called LINK-ports that implement point-to-point communications. In addition to TigerSHARC processors the accelerator comprises the control module. The control module is responsible for uploading programs to TigerSHARC processors, uploading and downloading information and controlling program start and stop processes on TigerSHARC processors. The control module communicates with the core processor (x86-based cores) via PCI bus. Typically, the first step at the stage of development of complex hardware platforms is the development of a computational node software model. For the purposes hereof there are virtual models considered as object models simulating the object behavior and obtained using specific software tools (simulator).

This article describes a method for developing computer simulation software tools and integrating the same with Sniper simulator [1]. Sniper simulator is designed for modeling multi-core systems based on x86-compatible cores featuring a complex cache memory hierarchy. The article explains how to integrate a simulation model of the accelerator into Sniper simulator and to develop and test the accelerator control driver based on the case study of the accelerator implementing a hash function computation algorithm.

"This work was supported by the Ministry of Education and Science of Russian Federation under contract No02.G25.31.0061 12/02/2013 (Government Regulation № 218 from 09/04/2010)"

II. Sniper Software Simulator and the Problem Set-Up

An essential advantage of Sniper simulator over other systems of similar purpose consists primarily in the capability of carrying out simulation modeling on distributed systems (clusters) which enables a considerable simulation speed-up.

The second essential advantage is the fact that Sniper simulator can be freely distributed in the form of source codes. This allows for integrating new functional capabilities into Sniper which is of fundamental importance as long as simulation of hardware accelerators is concerned. The basic part of Sniper is written in C++ but the number of simulator start-up and simulation data processing modules is implemented in Python. Sniper simulator features a modular structure wherein each

system element has its own description with the specified interface for interaction with other components. Each module can be adjusted by means of a set of parameters which values are specified in configuration files. In terms of implementation the simulator is a set of classes each of which is responsible for the specific architecture fragment or for back-up control elements.

Instead of x86 processor core simulator the system utilizes Pin software [2] developed by Intel specialists. Pin is a dynamic instrumentation tool. Dynamic instrumentation is a technology of embedding a complement code into the host program. This technology enables, firstly, to integrate a software performance test code and, secondly, to tap all software memory calls as well as syscalls. Tapping of memory calls allows for simulating a cache memory system with preset features and for creating a virtual common memory image for processes running on various cluster nodes. Simulation consists in execution of a multithreaded program and modeling of its operation on a multi-core architecture. Sniper links each application thread to the simulated architecture core. Each core being simulated may execute only one thread at a time, that is, the number of threads run by the rendering time system cannot exceed the number of cores configured in the target architecture. There are several processes run on one or more cluster nodes (each thread belongs to a process). Interprocess synchronization and data exchange is executed via TCP/IP protocol. The application threads are part of these processes and are controlled by the host operating system. Each thread has its own local timer and is synchronized with the other threads only when special events occur. A principal advantage of Sniper is the capability of simulating a multi-threaded application (that implies availability of a common memory) on a distributed independent memory system without a preliminary modification of the source code. Following the application simulation process Sniper generates a report with statistical data to enable performance analysis of the simulated system. Thus there is the number of executed commands, the number of cache hits and cache misses as well as some other parameters calculated for each core. This information is then used as a basis for the search for the optimum set of parameters by the user-defined scheme.

The key task of the study which results are set forth herein was to extend Sniper functions in order to enable performance analysis for computation systems which architectures include hardware accelerators. The first challenge within the study was to implement interfaces for integrating accelerator models into Sniper; another one was to develop a technology of transfer of C and C++ functions from the application level to the hardware accelerator model level. In addition, the study required to ensure that time delays caused by the hardware accelerator operation as such are accounted for.

III. Sniper Simulator with Cluster Node Accelerator Simulation Capabilities

In order to solve the problem set up there was a hardware accelerator connection interface as well as Linux-based accelerator driver emulation module implemented in Sniper [3]. Such an approach enables using Sniper simulator in development of drivers for hardware accelerators. There was Device class created for accelerator operation simulation purposes. The class encapsulates functionalities of the hardware accelerated emulated. This study offers a package of some digital processing functions implemented as a test model of accelerator. Each function has cluster run-time estimation functions implemented based on input data. The interaction between the accelerator and the application is carried out by means of system calls and the corresponding accelerator driver. The application initiates a system call (a standard syscall function of POSIX interface). The system call is tapped by Sniper simulator syscall processor. The processor includes a table of hardware accelerator drivers. If the processor determines that the syscall is addressed to a driver of the accelerator the corresponding driver function is called. At present Sniper supports three syscalls which are transmitted to the accelerator:

1) open: the system call enables access to the driver, calls dev_open driver function which initiates an accelerator model and recovers the driver descriptor for the application;
2) ioctl: syscall;
3) close: to release the device by the user application.

The driver processes system calls received from the application and controls the accelerator model. An accelerator model in Sniper is controlled by the driver by means of Device class function calls.

The set of macro definitions given in Appendix 1 allows for implementing a driver in such a way that the driver source code can be compiled unchanged both within Sniper and as a full core level driver for Linux operating systems. In order to ensure correct translation of system calls addressed to accelerator drivers there was Sniper simulator syscall processing unit modified. Specifically, the functional of the processor of the following system calls was changed:

1) open: opens a file or a device. In order to access the device in Linux OS the device must be opened by means of “open” syscall with the device name given for the argument. The modified Sniper simulator requires a unique name to be given to each hardware accelerator. If the name sent as an argument corresponds to the name of an accelerator Sniper will create a unique device descriptor, call dev_open driver function, and then recover the application descriptor. If the name sent as an argument does not correspond to the name of any accelerator Sniper will transmit the system call to the operating system.
2) ioctl: in our case ioctl is used to run a function of the accelerator. The device descriptor shall be sent as the first argument of the system call.
3) close: closes a file or a device. The device descriptor shall be sent as the only argument of this call. If the descriptor corresponds to the hardware accelerator model Sniper simulator shall run dev_close() function of the relevant driver and

assign the "closed" status to the device. After "close" syscall has been executed all other system calls with this descriptor shall be invalid.

By using the system call mechanism it is possible to start developing drivers and applications using the accelerator features before a physical accelerator appears.

IV. Conclusion

This study sets forth the results of a practically implemented approach to development of a toolkit for simulation modeling of computer systems of various architectures featuring hardware accelerators. The software tools developed increase the availability and performance of the processes of such systems development allowing for modeling the same as multipurpose in tasks and efficient in capacity.

References

[1] Carlson T. E., Heirman W., Eeckhout L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. Available: http://www.exascience.com/wp-content/uploads/2011/09/Sc2011carlson-final.pdf

[2] Reddy V., Settle A., Connors D. A., Cohn R. S. Pin: a binary instrumentation tool for computer architecture research and education. Available: http://www4.ncsu.edu/~efg/wcae/2004/submissions/connors.pdf

[3] J. Corbet, A. Rubini, G. Kroah-Hartman. Linux device drivers. O'REILLY – 2005.

Appendix A: Interface of class Driver

```
class DriverBase
{
private:
        DeviceBase* device;      // modeled device
        UInt32   device_id;
        String   device_name;

public:
   // initializing of Sniper system driver
   void setDriver( const UInt32 id, String dev_base_name);
   // returns device ID
   UInt32 getId( ) const;
   // returns device name
   String getName( ) const;

   // handler of syscall open
   virtual void dev_open();
   // handler of syscall close
   virtual void dev_close();
   // handler of syscall read
   virtual ssize_t dev_read( struct file*, char*, size_t, loff_t *);
   // handler of syscall write
   virtual ssize_t dev_write( struct file*, const char*, size_t, loff_t*);

   // handler of syscall ioctl
   virtual int dev_ioctl( unsigned int, unsigned long);
};
```

Appendix B: List of defines for modeling of accelerator drivers

```
#ifdef LINUX_DRIVERS_H
// Linux driver
#define drvdecl_func_read()  static ssize_t DEV_NAME##dev_read( struct file*, char*, size_t, loff_t*)
#define drvdecl_func_write()  static ssize_t DEV_NAME##dev_write( struct file*, const char*, size_t, loff_t*)
#define drvdecl_func_ioctl()  static long DEV_NAME##dev_ioctl(struct inode*, struct file*, unsigned int, unsigned long)
```



```
#define drvdecl_func_open()   static int DEV_NAME##dev_open( struct inode*, struct file*)
#define drvdecl_func_close()  static int DEV_NAME##dev_release (struct inode*, struct file*)
#define drvdecl_dev_start()   static int __init DEV_NAME##dev_init(void)
#define drvdecl_dev_finish()  static void DEV_NAME##dev_finish(void)

#define SET_DRV_MODULES() module_init(DEV_NAME##dev_init);          module_exit( DEV_NAME##dev_finish);

// set of defines, used by driver for device access
#define WRITE_REG(dev_addr,reg_id,value) *((int*)(dev_addr + reg_id)) = value
#define READ_REG(dev_addr,reg_id)        *((int*)(dev_addr + reg_id))
#define WRITE_MEM_PULL(device_addr,offset,host_addr,len) memcpy( ((char*)device_addr + offset, host_addr, len)
#define READ_MEM_PULL(device_addr,offset,host_addr,len)  memcpy( host_addr, ((char*)device_addr + offset, len)

#define print_log  printk
#define drv_malloc(size) kmalloc( size, GFP_KERNEL)
#define drv_free(size)

#else // driver as a part of Sniper

#define drvdecl_func_read()  ssize_t DEV_NAME::dev_read( struct file*, char*, size_t, loff_t*)
#define drvdecl_func_write() ssize_t DEV_NAME::dev_write( struct file*, const char*, size_t, loff_t*)
#define drvdecl_func_ioctl() long DEV_NAME::dev_ioctl( struct inode*, struct file*, unsigned int, unsigned long);
#define drvdecl_func_open()  int DEV_NAME::dev_open( struct inode*, struct file*)
#define drvdecl_func_close() int DEV_NAME::dev_release (struct inode *, struct file*)
#define drvdecl_dev_start()  DEV_NAME::dev_init(void)
#define drvdecl_dev_finish() DEV_NAME::~dev_finish(void)

#define SET_DRV_MODULES()

#define WRITE_REG(dev_addr,reg_offs,value) dev_addr->WriteReg(reg_offs,value)
#define READ_REG(dev_addr,reg_offs)      dev_addr->ReadReg(reg_offs)
#define WRITE_MEM_PULL(device,offset,host_addr,len) device->Copy2Dev(offset, host_addr, len)
#define READ_MEM_PULL(device,offset,host_addr,len)  device->Copy2Host(offset, host_addr, len)
#define print_log  printf
#define drv_malloc(size) malloc( size)

#endif
```

