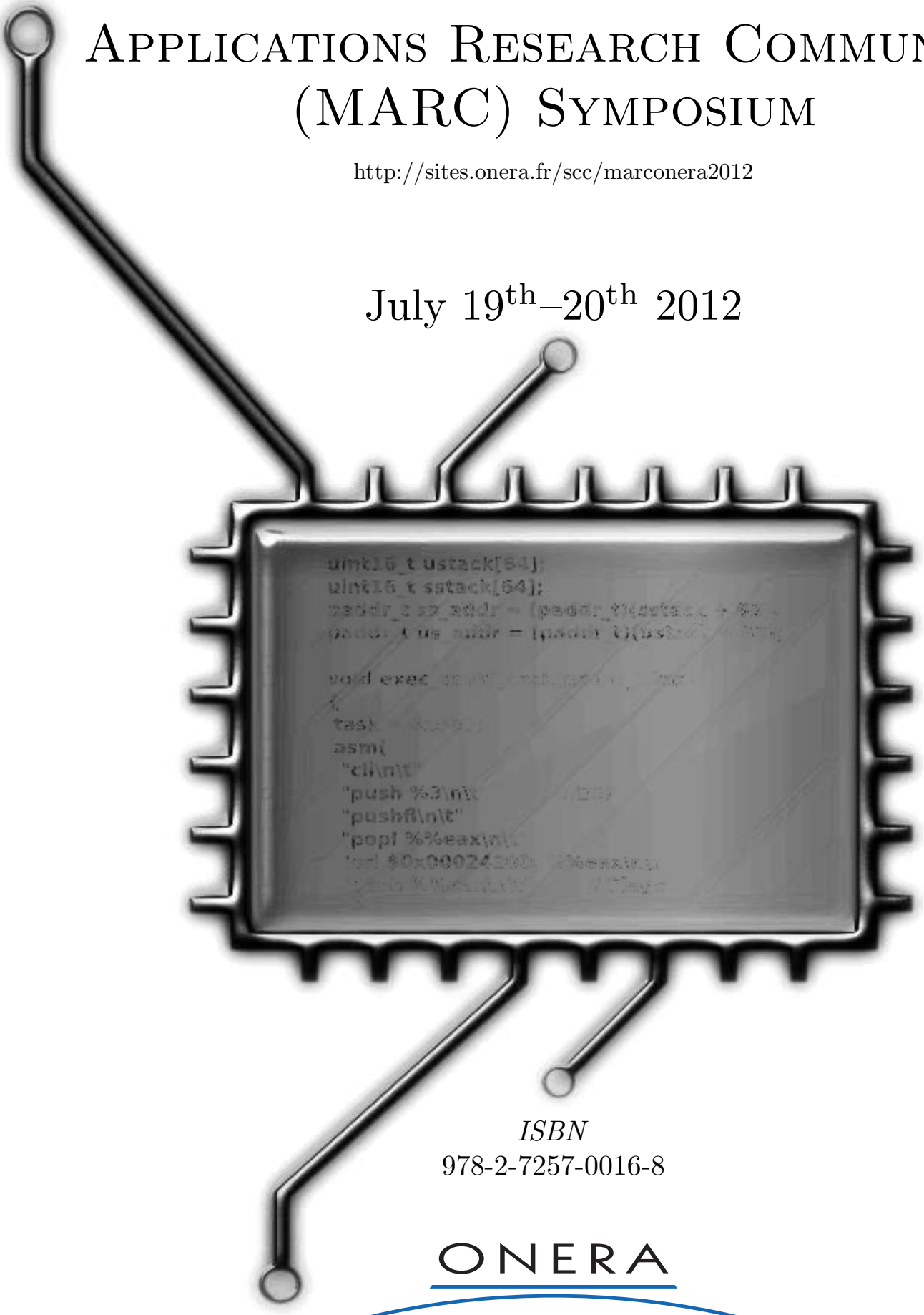
BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC

Michael Ziwisky, Dennis Brylow

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA
THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC

Michael Ziwisky
Department of Electrical and Computer Engineering
Marquette University
Milwaukee, WI 53233
Email: michael.ziwisky@mu.edu

Dennis Brylow
Department of Mathematics, Statistics, and Computer Science
Marquette University
Milwaukee, WI 53233
Email: brylow@mscs.mu.edu

***Abstract*—The many-core Intel SCC processor is one of a class of emerging, highly parallel computer architectures. Intel provides a modern Linux kernel which, running on the SCC as a separate instance per core, is able to load and launch user applications. However, there is a lack of open-source tools to facilitate development of "bare-metal" SCC applications – applications that are run directly on the chip without the support, overhead, or restrictiveness of an underlying operating system.**

To help fill this void, we present BareMichael, a minimalistic framework for compiling, loading, and launching mixed C and assembly code on the bare-metal Intel SCC. The framework also includes MikeTerm, a one-way pseudo-terminal for displaying output from an application as it executes on the chip. We share our solution in the hope that it will lower the barrier for others to begin development in a bare-metal environment on the SCC. Furthermore, we demonstrate the utility of BareMichael through two applications: supporting the use of the RCCE message-passing library, and serving as the foundation for a port of the Embedded Xinu operating system.

I. INTRODUCTION

The Single-Chip Cloud Computer (SCC) experimental processor is a "concept vehicle" created by Intel Labs as a platform for many-core software research [1], [2]. It features 48 processing cores based on the P54C architecture and a 256 Gb/s bisection bandwidth mesh network-on-chip (NoC). The chip is organized into 24 tiles, each of which contains two cores, a router, and 16 kB of shared memory that is accessible to all cores via the NoC. This fast, on-chip memory is referred to as the "message-passing buffer" (MPB).

Intel provides support software for SCC development including *SCC Linux*, a modern Linux kernel, and *sccKit*, a suite of tools for interacting with the chip via an attached "management console PC" (MCPC). While the environment of SCC Linux offers many convenient features, such as access to common Linux system services and the ability to interact with cores via an `ssh` session, it is also a restrictive environment, forcing developers to either run their SCC applications within a low CPU privilege level, or to modify the kernel itself to enable more advanced functionality.

It is thus desirable to be able to run applications in a "bare-metal" environment with neither the support nor the overhead and restrictions of an operating system. However, the barrier to get bare-metal C code running on the SCC and to get feedback from its execution is a significant one. We have overcome this barrier, and we share our solution, BareMichael, in the hopes that it will spare others the tedium and difficulty of coding the initialization and support routines necessary to begin development of bare-metal SCC applications.

The BareMichael framework enables a developer to execute bare-metal code on the SCC with supervisor-level access to all aspects of the chip. The framework is lightweight, minimalistic, and open-source. In the remainder of this paper, we describe the framework's platform initialization process, list the tools upon which it relies, describe a couple of applications for which we have used the framework, and discuss the other offerings for bare-metal SCC development.

II. THE BAREMICHAEL FRAMEWORK

BareMichael is a minimalistic framework to support bare-metal programming on the SCC. It is primarily a boot loader, not an operating system. Thus, it does not provide operating system functionality, but it may serve as a foundation upon which an operating system (or any other program) may be built. Along with a series of routines for initial configuration of an SCC core, BareMichael is packaged with `libxc`, a subset of the standard C library originally implemented for the Embedded Xinu kernel [3]. Upon the framework, developers may implement bare-metal code in C, x86 assembly, or a combination of the two. The framework also includes some SCC-specific helper functions and definitions to do things like reading the local core ID, reading mesh and tile clock frequencies, addressing MPBs and configuration registers, acquiring and releasing tile lock registers, and triggering inter-core interrupts. As BareMichael is an open-source tool, the implementations of all of these functions are exposed to the developer who is free to modify, remove, or reimplement them at will.

A. Platform Initialization

The following is a brief walkthrough of the code path BareMichael steps through to initialize an SCC core. This description, accurate for the latest versions (4, 5, 6, and 7) of the framework, illuminates the BareMichael startup process so that a developer may understand both how it works and

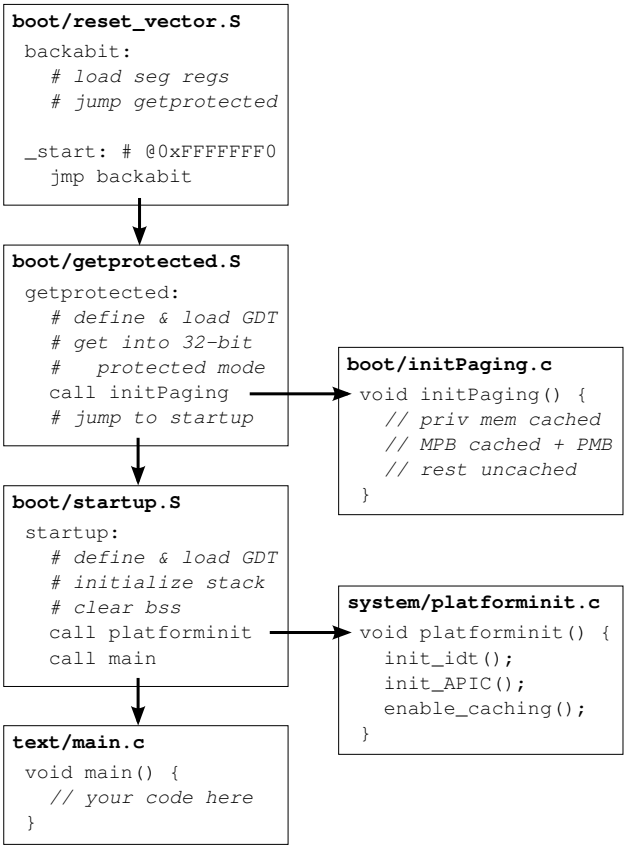


Fig. 1. Per-core initialization procedure of BareMichael.

how it may be modified to suit particular needs. Paragraph headers identify the location of the code being discussed, and a schematic representation of the entire process is illustrated in Figure 1.

a) boot/reset_vector.S: Based on the Intel P54C, each SCC core boots in "real mode," and consequently has access to just a 20-bit address space. In spite of this limitation, the first instruction a core executes after its reset pin is released is loaded from memory address 0xFFFFFFF0, sixteen bytes from the end of a 32-bit address space. We put a short relative jump instruction here, which takes us back just far enough to initialize the core's segment registers and stack pointer, then far-jump down to a `getprotected()` routine located within the first mebibyte of memory.

b) boot/getprotected.S: The `getprotected()` routine takes the processor into 32-bit "protected mode" by setting up the necessary CPU configuration data structures and registers, including a global descriptor table (GDT) to define flat code and data segments. Then a page table is created for virtual memory management.

c) boot/initPaging.c: The default look-up table (LUT) for an SCC core, which maps core addresses into a larger system address space, splits the core's address space into sections including private memory, shared memory, message passing buffer space, and configuration register space. Our page table flatly maps all of this space with cache disabled for all but private RAM and message passing buffers. Message passing buffer pages also have the PMB flag set to enable special caching features of the SCC [2]. With the page table configured and enabled, the core jumps to the `startup()` routine.

d) boot/startup.S: The `startup()` code gets linked together with `libxc` and the rest of the developer's bare-metal code to create the main image, which may be located in private memory wherever the developer chooses (specified via a Makefile variable). The `startup()` routine defines and loads a new (but identical) GDT within the main image to allow for easier addressing of the data structure should the developer wish to access it later. Space then is allocated for an interrupt descriptor table (IDT) which will be loaded with descriptors momentarily. After initializing a stack, clearing the `bss` section of the image, and initializing the floating point unit, the core calls `platforminit()`.

e) system/platforminit.c: Among the duties of the `platforminit()` routine are calls to initialize and enable the local advanced programmable interrupt controller (APIC), load the IDT with some default descriptors, and enable caching. As of version 3, the framework includes real-time clock support using the local APIC timer. If this feature is enabled (via a definition in `include/conf.h`), its initialization function is called here. Interrupt vectors 0x00 through 0x1F are reserved for CPU faults and exceptions, and the default handlers BareMichael assigns to these vectors print out information about the state that the system was in when the interrupt occurred. Such information is useful for debugging. After `platforminit()` returns, BareMichael calls the `main()` function in `text/main.c`, which is assumed to be the starting point of the developer's code.

To summarize, we now describe the state of an SCC core after BareMichael initialization. The setup routine brings the SCC core to 32-bit protected mode at privilege level 0 (supervisor level). Virtual memory management is enabled with page table entries present only for the core addresses that are mapped to actual system addresses by the default LUT configuration. Private memory is configured to have cache enabled, MPB-mapped pages have cache enabled and the SCC-specific PMB flag set, and all other sections have cache disabled. The local APIC is enabled and, by default, its periodic timer is set up to trigger a handler (found in `system/clock.c`) every millisecond. If the framework is configured for RCCE support (see Section III), the core's MPB space is initialized to zeros and a heap is initialized to allow dynamic management of private memory.

B. MikeTerm

BareMichael applications can print text back to the MCPC through a call to `printf()`. This function simply writes data to a circular buffer in memory where it can be seen and retrieved by the MCPC via the SCC's system interface. Each core has a different buffer allocated for this purpose. Running

```
[00]: Hello, World -- I'm core 0!

[01]: Hello, World -- I'm core 1!

[05]: Hello, World -- I'm core 5!

[24]: Hello, World -- I'm core 24!

[47]: Hello, World -- I'm core 47!

[00]: I'm going to trigger core 47's LINT0 now.

[47]: I've been interrupted!
[47]: (SCC has been booted for 2 seconds)

[00]: Now I'm toggling core 47's LINT1.

[47]: Another interruption!
[47]: (SCC has been booted for 5 seconds)
^C
Thanks for flying MikeTerm!
```

Fig. 2. Sample output from MikeTerm. In this sample program, each booted core says "Hello." Then, after a short delay, core 0 toggles each of core 47's APIC interrupt pins with a delay in between. Core 47 has set these interrupt vectors to point to handlers that print out the total time passed since boot up. That time is kept track of by the real-time clock which operates based on the APIC timer and the tile clock frequency.

on the MCPC, a utility called *MikeTerm* acts as a one-way pseudo terminal, periodically polling each of the 48 buffers and printing any text found therein. All output from MikeTerm is preceded by a core identifier. Because MikeTerm scans the shared memory buffers sequentially, it is not guaranteed that its output will be printed in the order in which the cores wrote to their respective buffers. The output from any given core will be delivered in the order in which the core printed it, but ordering of output between any two cores is not necessarily preserved. Additionally, if a core is writing to its buffer faster than MikeTerm is retrieving it, old data will be overwritten and lost without being printed. No protections are built in to prevent this. The default configuration of the framework allocates 64 KiB buffers which get polled by MikeTerm roughly once per second, so data is likely to be lost when output rates are greater than about 64,000 characters per second. BareMichael currently offers no mechanism for interacting with running SCC programs by feeding data in the other direction, from the MCPC to the chip.

C. Build Environment and Dependencies

1) Dependencies: BareMichael leverages some open-source utilities for image compilation, image loading, and delivering output through MikeTerm. The framework uses the `i386-unknown-linux-gnu` cross-compiler tools from gcc version 3.4.5 to produce flat binary object files. *sccKit* is a suite of utilities, provided by Intel, that run on the MCPC and interact with the SCC. BareMichael is compatible with sccKit version 1.4.1, and it uses the `bin2obj`, `sccMerge`, `sccBoot`, and `sccReset` tools for loading binaries into SCC memory and toggling reset pins of individual cores. MikeTerm uses `sccDump` and `sccWrite` to access print buffers in shared memory.

2) Compilation and Execution: Compilation of both MikeTerm and the SCC image is managed using Makefiles written for the GNU `make` utility. MikeTerm is written in C++ and located in the `miketerm` directory. To compile it, simply change to that directory and invoke `make`.

BareMichael expects the directory containing sccKit binaries to be included in the user's `PATH` environment variable. Paths to the cross-compiler and `bin2obj` tool must be specified in the framework's Makefile, which is located at `compile/Makefile`. The Makefile also includes a configuration variable for specifying a list of cores to boot. After defining these few variables, compiling and running a bare-metal application is very simple and straightforward. The default `make` target builds the image; the `run` target loads that image into SCC memory and releases the resets of the specified cores. The `main()` function in `test/main.c` is the entry point for the developer's code, and if all of the developer's code is contained in that file (or in any set of files already in the framework), a simple '`make; make run`' is all that is needed to get the code running on the SCC. Follow it up with '`../miketerm/miketerm`' to view output from the cores. If additional source files need to be linked, one must add them to one of two lists in the Makefile: C source files get added to the `C_FILES` list, while assembly files belong in the `S_FILES` list.

3) Advanced Capabilities: Though most developers probably will be satisfied with the default configuration of the build environment, additional customization is possible. One simple example is changing the memory address to which the main image gets loaded onto the core. This is easy to modify as it is already defined by a variable (`IMG_ADDR`) in the Makefile. However, the framework has other potential capabilities – such as loading and booting different images on different cores – that are possible to realize but not as simple to exploit. For this reason, we disclose the roles of a few files that the build process creates along the way to creating a loadable SCC image.

Initially, the source is compiled into three flat binary object files: the reset vector, the "get protected" and paging initialization code, and the main image. The file `compile/load.map` is created and populated with the names of these three objects, each preceded by the memory address (32-bit core address, not a memory controller address) to which it is to be loaded. This file serves as the input to the `bin2obj` tool, which creates a text file, `compile/battle.obj`, that represents a composite of the three objects. The `sccMerge` tool decides where to load the composite image into SCC memory and how to set initial core LUT configurations. The tool makes these decisions based on three arguments: the number of cores to be served by each memory controller (12 by default), the size per memory controller in GiB (8 by default), and the contents of a `.mt` input file. BareMichael creates the file `compile/battle.mt`

and populates it with 48 lines, each of which identifies a core, a memory controller, a "memory slot" (between 0 and 47, inclusive), and a `.obj` file. By default, this file assigns to each core: the nearest memory controller; a memory slot between 0 and 12, which is assigned in increasing numerical order to the 12 cores sharing a memory controller; and the object file that was built earlier, `compile/battle.obj`. The output of `sccMerge` is a directory, `compile/obj/`, and files therein that define the SCC memory contents and the LUT configurations. This directory is provided as an argument to `sccBoot`, which does the actual loading of SCC memory and configuring of LUTs. Finally, the framework issues the command `'sccReset -r <list of cores>'` to release the reset pins of the desired cores.

Clearly, the build procedure may be altered in a few ways – most notably through modifications to the `.mt` file – to customize how SCC memory gets loaded and distributed among cores. As an example, one may arbitrarily assign `.obj` files to cores in the `.mt` file to boot heterogeneous images among the cores. Of course, this requires building multiple `.obj` images, so multiple load maps must be defined and fed to `bin2obj`. Implementation of such alterations is left to the interested developer.

III. Integration with RCCE

RCCE [4] is a message-passing software library that Intel Labs designed and implemented in conjunction with the SCC hardware. The current version of the library, V2.0, may be compiled for use in SCC Linux, a kernel port also supplied by Intel, or for use in a bare-metal environment. However, because bare-metal RCCE is a library and not an environment itself, it does not provide the execution framework needed to run bare-metal applications on its own. In addition to a CPU initialization process, RCCE demands:

- POSIX functions `mmap()` and `munmap()` for virtual memory management,
- file operations such as `open()`, `flush()`, and `fprintf()`,
- `malloc()` and `free()` for dynamic memory management, and
- various additional C library functions.

These gaps are filled by the v6 release of BareMichael, allowing the developer to use the unmodified bare-metal RCCE library with BareMichael "out of the box." While some features such as dynamic memory management are properly implemented for general use, others, including virtual memory management functions and file operations, are tailored to be compatible with RCCE, though not fully implemented to fulfil their intended duties. These functions are not necessarily safe for use outside of the purpose of supporting RCCE V2.0.

We now present some performance results for RCCE V2.0 running in the BareMichael environment. The simple "ping-pong" benchmark [5] was run on cores 0 and 1 with the SCC mesh and memory running at 800 MHz and core clocks of 533 MHz. As seen in Figure 3, the benchmark exhibits nearly identical performance regardless of whether it is run within

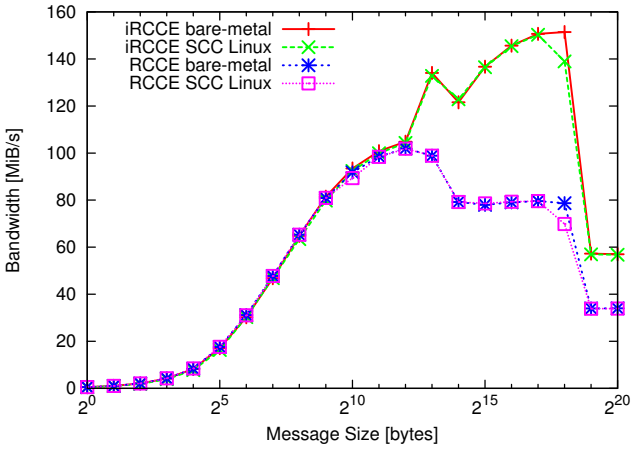


Fig. 3. Ping-pong benchmark results for RCCE and iRCCE running in SCC Linux and BareMichael environments.

SCC Linux or BareMichael. The same is seen when running the benchmark with the pipelining send and receive functions of iRCCE V1.2 [6] in both environments.

IV. Implementation of Xipx OS

Due to its minimalistic nature, BareMichael is a suitable foundation not only for running individual parallel applications, but also for launching operating system kernels. We demonstrate this with *Xipx*, an SCC port of the Embedded Xinu operating system that leverages the BareMichael framework for hardware initialization [7]. The following section presents the Xipx MPB device, which stands as an asynchronous alternative to the RCCE/iRCCE way of managing the SCC's message passing hardware.

A. The Xipx MPB Device

Xipx exposes the SCC message passing buffers via the standard Xinu device API [8]. Several instances of an MPB device are created at boot time, and each one acts as a two-way message passing channel. As an asynchronous and interrupt-driven driver, the Xipx MPB device facilitates inter-core communications in a way that is fundamentally different than RCCE. The basic RCCE API uses a symmetric name space model, meaning all cores access shared variables in the same way – using a variable name and the core ID of the MPB where the variable is stored. In order to preserve this symmetry, certain RCCE routines must be encountered jointly by all cores involved in the system. These routines are referred to as "collective operations," and saying they are "encountered jointly" means that they get called in the same order with respect to each other on all cores in the system. For example, the `RCCE_malloc(size)` routine, which allocates `size` bytes in the local MPB, is a collective operation – any core calling `RCCE_malloc(size)` is counting on all other cores to do the same in the same order with respect to other collective operations. This ensures that all cores are returned a pointer with the same offset from the beginning of their

 ISBN: 978-2-7257-0016-8

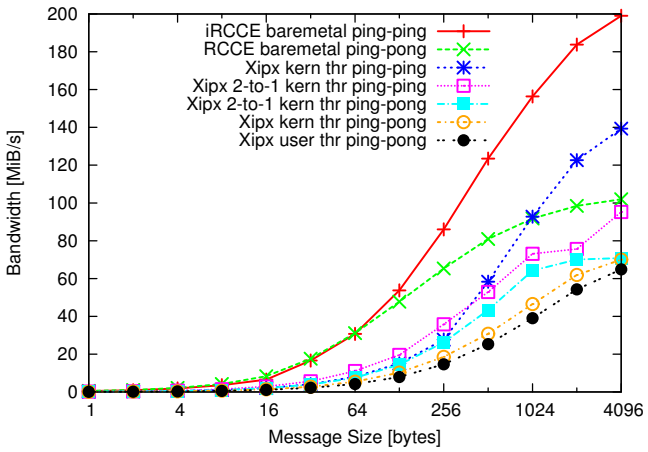


Fig. 4. Benchmark performance of the asynchronous Xipx MPB device.

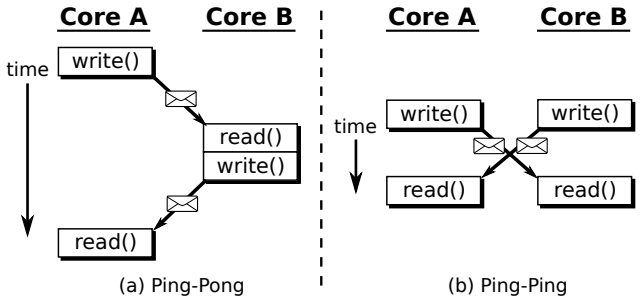


Fig. 5. Comparison of the communication patterns for (a) the ping-pong benchmark and (b) the ping-ping benchmark.

respective MPBs, and they can therefore safely assume the correct location of the corresponding variable in any other remote MPB.

The symmetric name space of a RCCE application is a convenient and efficient way to manage MPB space for a single application, but it is not capable of supporting multiple simultaneous applications on the SCC. With multiple parallel applications running at a time, one cannot guarantee the order in which the applications will get CPU time on each core, and therefore cannot guarantee that collective operations are encountered jointly by all cores. As a simple example, consider two different applications running on the SCC. One of them runs on cores 0 and 1, the other on cores 0 and 2. Because core 0 is involved in both applications, but cores 1 and 2 are only involved in one each, any collective operation core 0 performs in one application is not performed by its communicating partner in the other, therefore the name space symmetry is broken.

In order to support an arbitrary graph of communicating threads on SCC cores, the Xipx MPB device does not assume a symmetric name space. Instead, Xipx treats each MPB as a FIFO buffer. Messages are written to the receiving core's MPB with a header to indicate the core and channel from which it was sent, the channel to which it should be delivered, and the length of the payload. These messages can arrive from any core in any order, and the presence of a new message is signalled by an interrupt. The handler for this interrupt searches through the local MPB devices to find one that is open on the channel indicated by the message header. It then copies the message to a pre-allocated buffer and sends a signal to the thread that owns the device so that a subsequent (or pending) call to `read()` will retrieve the data.

Figure 4 illustrates the performance of the Xipx MPB device for a number of scenarios. All benchmarks were run with the same hardware configuration as described in Section III. In addition to the basic ping-pong benchmark, we executed the "ping-*ping*" benchmark in which two cores each simultaneously send a message to each other and then simultaneously retrieve the message they were sent. The communication patterns of the ping-pong and ping-ping benchmarks are illustrated in Figure 5. In a two-to-one ping-X test, one core runs two simultaneous ping-X benchmarks, each with a different partner core. The measured bandwidth is total data flow in and out of the shared core. For comparison, the RCCE ping-pong performance is duplicated here and iRCCE ping-ping data is introduced.

The Xipx MPB device achieves about 70% of the bandwidth of RCCE and iRCCE in ping-pong and ping-ping benchmarks, respectively. Xipx kernel threads slightly outperform user threads due to the overhead associated with user thread system calls. Though the Xipx device does not match the two libraries in raw bandwidth, the two-to-one benchmarks it performs are not even possible with those libraries. As we have already discussed, this is because the collective communications on which the libraries rely prohibit their use in two concurrent programs. Furthermore, the absence of a RCCE ping-ping benchmark is due to the fact that the library's synchronous semantics render it incapable of implementing that communication pattern.

B. *Porting an OS with BareMichael*

Xipx is, in fact, the precursor of BareMichael; the framework was extracted from Xipx as the initial set of operations that set up a C execution environment in 32-bit protected mode. Due to this development history, the authors cannot comment on the effort required to port another x86-based OS to the SCC using BareMichael as an aide. However, Xipx diverges from BareMichael beginning in the `startup()` routine, and we believe that the execution path preceding that point is generic enough to be useful for other operating systems as hardware initialization code. The `initPaging()` routine may be replaced or modified to set up an appropriate initial pagetable. Furthermore, regardless of the build process used to generate the OS image, the build environment of BareMichael should be useful for merging that image with the framework's initialization code and loading the resulting composite image into SCC memory.

V. RELATED WORK

Microsoft has released a Visual Studio add-in and bare-metal environment package for the SCC [9]. The source

code for the minimalistic bare-metal environment is provided, thereby allowing developers to modify the environment to suit their needs. However, development options are limited as the Microsoft tools must be run from a Windows machine that has a network connection to the MCPC. Furthermore, the license for this framework allows for non-commercial use only, and it grants back to Microsoft the right to use, modify, and sell any modifications to and/or derivative works of their framework. BareMichael tools are run directly on the MCPC, and its open-source, BSD-style license is less restrictive, permitting redistribution and use of the framework and derivative works, both in source and binary forms, for both commercial and non-commercial purposes.

ETI provides a beta version of its SCC Development Framework [10] for compiling and launching bare-metal applications on the SCC. Applications get compiled into an ELF format binary and are loaded and launched via a utility running on the MCPC. The ETI framework is closed source, and therefore lacks certain flexibilities offered by BareMichael such as the ability to modify the boot process. The current release also offers no means of specifying which cores to boot up, only the number of cores. Furthermore, it forces the same image to be loaded onto all cores at once. In contrast, BareMichael is able to load different images to different cores with only minor changes to the build process.

Finally, an Intel internal framework named *BareMetalC* exists [5], but it is not released to the public due to licensing limitations. The bare-metal RCCE library was created specifically to support this framework.

VI. CONCLUSION

We have introduced BareMichael, a minimalistic, open-source framework for loading and executing bare-metal programs on the Intel SCC architecture. Our lightweight framework is packaged with a subset of the standard C library, and features out-of-the-box support for Intel's message-passing library, RCCE. A basic benchmark shows that message-passing bandwidth for RCCE on bare-metal is nearly identical to that for RCCE in SCC Linux.

A programmer developing in BareMichael is not limited merely to launching individual parallel applications on the SCC. The flexibility of the framework is demonstrated by our implementation of Xipx, a port of the Embedded Xinu operating system, for which BareMichael serves as the foundation. In order to allow multiple threads to simultaneously use the SCC's message passing buffer in a preemptive environment, Xipx manages the MPB hardware at the device layer. Though our simple device implementation does not match the bandwidth of RCCE, it allows for asynchronous communications and allows multiple processes to use the MPB simultaneously, two features that are not possible with the basic RCCE API. Future work on Xipx will investigate how to increase message-passing performance at the device layer.

Typical usage of the SCC involves loading the private memory of each core with an identical image. However, there is interest in being able to boot different images on different cores [11], [12]. The utilities of sccKit allow for this, and we have been successful in using the BareMichael build environment to load and boot heterogeneous images on the SCC. A future release will incorporate this functionality.

The current, simplistic implementation of MikeTerm only allows one-way serial communication from the SCC to the MCPC. Two-way communication is desirable, and it may be realized via the UART support that was introduced with version 1.4.2 of sccKit. We plan to look into this possibility for future releases as well.

We provide the BareMichael framework as an open-source package in the hope that it will lower the entry barrier for others wishing to develop and run bare-metal applications on the Intel SCC. The framework is available for download at http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/.

ACKNOWLEDGMENT

The authors would like to thank the members of the Intel Many-core Applications Research Community – in particular Ted Kubaska and Jan-Arne Sobania – for their prompt and clear responses to questions arising during development on the SCC. Thanks also to Intel Corporation for access to the SCC hardware.

REFERENCES

[1] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. R. M. Gries, G.Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. V. der Wijngaart, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[2] *SCC External Architecture Specification (EAS)*, Intel Corporation, Nov. 2010, revision 1.1.

[3] D. Brylow and B. Ramamurthy, "Nexos: A next generation embedded systems laboratory," *SIGBED Review*, vol. 6, no. 1, Jan. 2009, URL http://sigbed.seas.upenn.edu/.

[4] T. Mattson and R. van der Wijngaart, "RCCE: A small library for many-core communication," Intel Corporation, Jan. 2011, software Version 2.0-release.

[5] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC processor: The programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[6] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, Jul. 2011, pp. 525–532.

[7] M. W. Ziwisky, "A message-passing, thread-migrating operating system for a non-cache-coherent many-core architecture," Master's thesis, Marquette University, to be published.

[8] D. E. Comer, *Operating System Design: The XINU Approach, Linksys Version*. CRC Press, 2011.

[9] (2011, Mar.) Visual Studio add-in and bare-metal environment for Intel SCC. Microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/downloads/37ccb116-c67d-4c44-9181-898889b8352d/

[10] (2011, Aug.) ETI's SCC development framework available. Intel MARC forums. [Online]. Available: http://communities.intel.com/thread/17643/

[11] (2011, Jun.) Booting custom kernels on the SCC. Intel MARC forums. [Online]. Available: http://communities.intel.com/message/128484/

[12] (2011, Sep.) rccerun to start different applications on cpu cores. Intel MARC forums. [Online]. Available: http://communities.intel.com/message/137635/