Performance and Power Analysis of RCCE Message Passing on the Intel Single-Chip Cloud Computer

John-Nicholas Furst Ayse K. Coskun
Electrical and Computer Engineering Department, Boston University, Boston, MA 02215 USA
{jnfurst, acoskun}@bu.edu

***Abstract*— The number of cores integrated on a single chip increases with each generation of computers. Traditionally, a single operating system (OS) manages all the cores and resource allocation on a multicore chip. Intel's Single-chip Cloud Computer (SCC), a manycore processor built for research use with 48 cores, is an implementation of a "cluster-on-chip" architecture. That is, the SCC can be configured to run one OS instance per core by partitioning shared main memory. As opposed to the commonly used shared memory communication between the cores, SCC cores use message passing. Intel provides a customized programming library for the SCC, called RCCE, that allows for fast message passing between the cores. RCCE operates as an application programming interface (API) with techniques based on the well-established message passing interface (MPI). The use of MPI in a large manycore system is expected to change the performance-power trends considerably compared to today's commercial multicore systems. This paper details our experiences gained while developing the system monitoring software and benchmarks specifically targeted at investigating the impact of message passing on performance and power of the SCC. Our experimental results quantify the overhead of logging messages, the impact of local versus global communication patterns, and the tradeoffs created by various levels of message passing and memory access frequencies.**

I. Introduction

Processor development has moved towards manycore architectures in recent years. The general trend is to utilize advances in process technology to include higher numbers of simpler, lower power cores on a single die compared to the previous trend of integrating only a few cores of higher complexity. This trend towards integrating a higher number of cores can be seen in desktops, servers, embedded platforms, and high performance computing (HPC) systems. Future manycore chips are expected to contain dozens or hundreds of cores.

While integrating a high number of cores offers the potential to dramatically increase system throughput per watt, manycore systems bring new challenges, such as developing efficient mechanisms for inter-core communication, creating strategies to overcome the memory latency limitations, and designing new performance/power management methods to optimize manycore system execution. A significant difference of manycore systems compared to current multicore chips comes from the on-chip communication: manycore systems are likely to incorporate a network-on-chip (NoC) instead of a shared bus to avoid severe performance limitations. One method of enabling inter-core communication on a NoC is a message passing interface (MPI).

In order to enable new research in the area of manycore design and programming, Intel Labs created a new experimental processor. This processor, called the "Single-Chip Cloud Computer" (SCC), has 48 cores with x86 architecture. The SCC chip provides a mesh network to connect the cores and four memory controllers to regulate access to the main memory [5]. The SCC includes an on-chip message passing application framework, named RCCE, that closely resembles MPI. RCCE provides multiple levels of interfaces for application programmers along with power management and other additional management features for the SCC [9].

The objective of this paper is to investigate the on-die message passing provided by RCCE with respect to performance and power. To enable this study, we first develop the monitoring tools and benchmarks. Our monitoring infrastructure is capable of logging messages, track performance traces of applications at the core level, and measure chip power simultaneously. We use this infrastructure in a set of experiments quantifying the impact of message traffic on performance and power. Significant findings of this paper are: overhead of our message logging method is negligible; execution times of applications increase with larger distances between communicating cores; and observing both the messages and the memory access traffic is needed to predict performance-power trends.

We present the monitoring infrastructure for the SCC in Section II. Section III describes the applications we developed for SCC. Section IV documents the experimental results on message logging overhead, effects of various message/memory access patterns, and energy efficiency. Section V discusses related work. Section VI concludes the paper.

II. Monitoring Infrastructure for the SCC

Analyzing the message passing system on the SCC requires monitoring performance and power consumption of the system at runtime. As the SCC was designed as a research system it includes special hardware and software features that are not typically found in off-the-shelf multi-core processors. Additional infrastructure is required to enable accurate and low-cost runtime monitoring. This section discusses the relevant features in the SCC architecture and provides the details of the novel monitoring framework we have developed.

Hardware and Software Architecture of the SCC:

The SCC has 24 dual-core tiles arranged in a 6x4 mesh.

Each core is a P54C CPU and runs an instance of Linux 2.6.38 kernel. Each instance of Linux executes independently and the cores communicate through a network interface. Each core has private L1 and L2 caches. Cache coherence is managed through a software protocol as opposed to commonly used hardware protocols. Each tile has a message passing buffer (MPB), which controls the message exchange among the cores. The SCC is connected by a PCI-Express cable to a PC acting as the Management Console (MCPC).

The SCC system has a power sensor used for measuring the full SCC chip power consumption. Power is measured by polling this sensor during application execution.

Each P54C core has two performance counters. These counters can be programmed to track various architectural events, such as number of instructions, cache misses or memory accesses. Performance counters can be accessed from the core in which they are located by reading dedicated registers.

The SCC software includes RCCE, which is a lightweight message passing library developed by Intel and optimized for SCC [9]. It uses the hardware MPB to send and receive messages. At the lower layer, the library implements two message passing primitives *RCCE_put* and *RCCE_get*. These primitives move the data between a local core buffer to the MPB of another core.

Our system setup includes the SCC, the MCPC, and the monitoring framework we developed. On the SCC we implemented utilities to track performance counters, collect power measurements, and log message traffic. On the MCPC we developed software to load desired benchmarks and experimental configurations to the SCC. After running experiments, we analyze the collected data using our automated software.

Software Modules Developed for the SCC:

Performance Counters: To implement performance counter polling we added a patch to the Linux kernel that RCKOS runs. RCKOS is the operating system the SCC cores run and is provided by Intel. Our patch involves a device used for polling and the `ioctl` infrastructure to communicate with the device. The `ioctl` is a system call that is device-specific and allows for user-land access to protected kernel functionality. Once the kernel is patched, RCKOS is recompiled and a new binary image is created and used to flash the cores.

Performance polling is performed by setting the Control and Event Select Register (`CESR`). The P54C cores on the SCC have two registers (0x12, 0x13) allotted for performance counters. The `CESR` contains a 6-bit Event Select field (ES), a Pin Control bit (PC) and a three bit control field (CC) for each of the two counters. The `CESR` is located at `0x11` and is visualized in Figure 1. Two independent events can can be counted by setting the appropriate codes for each ES. The Counter Control is used for enabling / disabling the counters. The `CESR` is programmed through using the Model Specific Registers (MSR) which are available on RCKOS through the `/dev/msr0` device.

To measure L1 cache misses and instructions, we wrote *0xD600CE* to the MSR. For tracking memory access density,

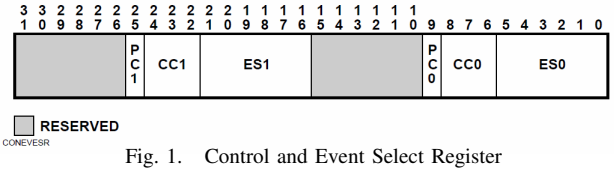


Fig. 1. Control and Event Select Register

we poll the counter "non-cacheable memory reads." We empirically determined non-cacheable memory reads as a good metric for quantifying memory access intensity through measurements with a set of custom designed microbenchmarks that vary in their memory access density [6]. To measure memory accesses and instructions, we wrote *0xD600DE* to the MSR. We ran multiple experiments to collect all three parameters (cache misses, memory accesses, number of instructions). It is also possible to multiplex the register polling to increase the number of parameters collected with little loss of accuracy.

Message Logger: We modified the lower level *RCCE_put* and *RCCE_get* routines in the RCCE library to log the number of messages sent and the source/destination of each message. At the end of each parallel thread the library generates a log containing the communication matrix. Each element in the matrix $\{m_{i,j}\}$ corresponds to the number of messages that $core_i$ has sent to $core_j$. In addition, we program the RCCE library to trigger the logging of the performance counters at the beginning of each of the parallel threads and save the trace at the end of execution.

Software Modules Developed for the MCPC:

- *Stress files and app-loader*: These files contain the benchmark sequences for the tests. For each benchmark, the stress file provides the name, number of threads, and the cores to allocate the benchmark. The *app-loader* loads the files on the SCC to start the experiments. We wrote a set of python scripts that run on the MCPC. These scripts load the stress configuration files and start the RCCE benchmarks in SCC.
- *Post-processing SW*: We designed software for processing the collected data. This script interfaces with the collected data and the stress file. For each benchmark, the script collects the logs and parses them to extract useful statistics. The script then stores the parsed measurements in a MySQL database stored on the MCPC. A custom web-based front end to this database was created to display the results. The data are available for access by Matlab or Excel allowing the implementation of other complex analysis functions.

In this paper we use the monitoring infrastructure described above for analyzing the message passing system on the SCC. The framework can also be leveraged for enabling runtime management policies on the SCC computer.

III. Application Space

We employ a set of benchmarks to evaluate the performance of the SCC and explore a variety of configurations. Two of these benchmarks are Block Tridiagonal (*BT*) and Lower-Upper (*LU*) from the NAS parallel benchmarks (NPB) suite [1]. *BT* and *LU* have been re-programmed for the Intel SCC, and are available to the MARC community. We also

use other benchmarks provided by Intel for the SCC. We build upon the existing benchmarks to create a wider set of operating scenarios in terms of number of cores used and the message traffic. We also design a broadcast benchmark to emulate one to multiple core communication. The complete benchmark set we run in our experiments is as follows.

Benchmarks provided by Intel:

- *BT*: Solves nonlinear Partial Differential Equations (PDE) with the Block Tridiagonal method.
- *LU*: Solves nonlinear PDEs with the Lower-Upper symmetric Gauss-Seidel method.
- *Share*: Tests the off-chip shared memory access.
- *Shift*: Passes messages around a logical ring of cores.
- *Stencil*: Solves a simple PDE with a basic stencil code.
- *Pingpong*: Bounces messages between a pair of cores.

Custom-designed microbenchmark:

- *Bcast*: Sends messages from one core to multiple cores.

The broadcast benchmark, *Bcast*, sends messages from a single core to multiple cores through RCCE. We created the benchmark based on the *Pingpong* benchmark, which is used for testing the communication latency between pairs of cores using a variety of message sizes.

Table I categorizes the Intel benchmarks based on instructions-per-cycle (IPC), Level 1 instruction (code) misses (L1CM), number of messages (Msgs), execution time in seconds, and memory access intensity. All parameters are normalized with respect to 100 million instructions for a fair comparison. Each benchmark in this categorization runs on two neighbor cores on the SCC. The table shows that the *Share* benchmark does not have messages and is an example of a memory-bounded application. *Shift* models a message intensive application and *Stencil* models an IPC heavy application. *Pingpong* has low IPC but heavy L1 cache misses. *BT* has a medium value for all performance values except for the number of messages. *LU* is similar to *BT* except that it has even higher number of messages and the lowest number of L1 code cache misses.

We update the *Stencil*, *Shift*, *Share*, and *Pingpong* benchmarks so that they can run on cores in configurations determining which cores communicate and which cores are utilized. Note that for all configurations of these benchmarks, communication occurs within "pairs" of cores (i.e., a core only communicates to a specific core and to no other cores). The configurations we used in our experiments are as follows:

- *Distance between the two threads in a "pair"*:
  - *0-hops*: Cores on the same tile (e.g., cores 0 and 1)
  - *1-hop*: Cores on neighboring tiles (e.g., cores 0 and 2)
  - *2-hops*: Cores on tiles that are at 2-hops distance (e.g., cores 0 and 4)
  - *3-hops*: Cores on tiles that are at 3-hops distance (e.g., cores 0 and 6)
  - *8-hops*: Cores on corners (e.g., cores 0 and 47)
- Parallel execution settings:
  - *1 pair*: Two cores running, 46 cores idle
  - *2 pairs*: Four cores running, 44 cores idle
  - *3 pairs*: Six cores running, 42 cores idle
  - *4 pairs*: Eight cores running, 40 cores idle
  - *5 pairs*: Ten cores running, 38 cores idle
  - *6 pairs*: Twelve cores running, 36 cores idle
  - *24 pairs*: 48 cores running

  The idle cores run *SCC Linux* but do not run any user applications and they are not in sleep states.
- *Broadcast*: The *Bcast* benchmark is run with one core communicating to N cores, where $1 \leq N \leq 47$.

TABLE I. BENCHMARK CATEGORIZATION. VALUES ARE NORMALIZED TO 100 MILLION INSTRUCTIONS.

Benchmark	L1CM	Time	Msgs	IPC	Mem.Access
Share	High	High	Low	Low	High
Shift	High	Low	High	Medium	Low
Stencil	Low	Low	Low	High	Medium
Pingpong	High	Medium	Medium	Low	Low
BT.W.16	Medium	Medium	High	Medium	Medium
LU.W.16	Low	Medium	High	Medium	Medium

Benchmark Categorization (normalized to 100M inst)—*Numerical*					
Benchmark	L1CM	Time	Msgs	IPC	Mem.Access
Share	372361	3.3622	871	0.0558	0.05
Shift	307524	0.7784	147904	0.2410	0.001
Stencil	97715	0.5528	23283	0.3393	0.03
Pingpong	280112	2.1116	68407	0.0888	0.001
BT.W.16	251096	1.11	229411	0.1682	0.03
LU.W.16	94880	1.15	305988	0.1631	0.03

The applications were run 5 times and the collected data have been averaged. An additional warmup run was conducted before the experimental runs. All of the experiments were conducted with the tiles at 533 MHz, the mesh at 800MHz and the DDR's at 800MHz. Our recent work also investigates the impact of frequency scaling on the SCC power and performance [2].

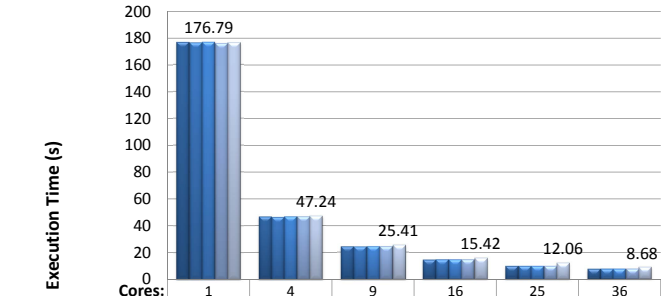
IV. EXPERIMENTAL EVALUATION

The purpose of the experiments is to quantify the performance and power of the Intel SCC system while running applications that differ in number of messages, message traffic patterns, core IPC, and memory access patterns. In this way, we hope to understand the performance-energy tradeoffs imposed by using MPI on a large manycore chip.

A. Overhead of Message Logging

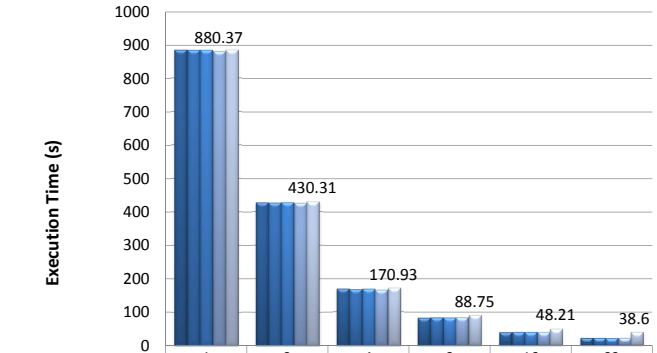
We first analyze the overhead caused by our message logging and performance monitoring infrastructure. Figures 2 and 3 demonstrate the overhead measured in execution time caused by different levels of measurement while running *BT* and *LU*. We choose *BT* and *LU* to study message over logging overhead as they are standard multicore MPI benchmarks. In the figures, *control* represents the case without any logging, *performance counters* results are for tracking performance counters only, *counting messages* is for logging both counters and number of messages, *message target* also logs the sender/receiver cores for each message, and *message size* logs the size of each message on top of all the other information.

We see in figures 2 and 3 respectively that while there is an overhead associated with the message logging, it is very small.



Cores:	1	4	9	16	25	36
Control	176.76	46.67	24.32	14.14	9.81	7.37
Performance Counters	176.39	46.36	24.23	14.24	9.81	7.45
Counting Messages	176.79	46.79	24.39	14.28	9.8	7.43
Message Target	175.82	46.8	24.43	14.24	9.77	7.47
Message Size	176.45	47.24	25.41	15.42	12.06	8.68

Fig. 2. *BT* Class W Execution Time(s) vs. # of Cores vs. level of logging. The execution time is shown for a varying number of cores. In each case the the addition of logging shows very small overhead.



Cores:	1	2	4	8	16	32
Control	884.43	427.94	168.24	80.87	38.72	21.48
Performance Counters	883.98	427.32	166.81	81.85	39.27	21.54
Counting Messages	884.04	428.18	167.92	81.51	39.37	21.3
Message Target	880.37	426.68	164.96	81.99	39.12	21.55
Message Size	884.42	430.31	170.93	88.75	48.21	38.6

Fig. 3. *LU* Class W Execution Time(s) vs. # of Cores vs. level of logging. The execution time is shown for a varying number of cores. Again, the overhead of message logging is very low.

We have seen similar low overhead when we measured the overhead for the other benchmarks. For example, for *message target* logging, we see only a 0.21% overhead in execution time when running the *Stencil* benchmark.

When logging message size is added to the infrastructure we see a significant increase in execution time, especially when a large number of cores are active (see Figures 2-3). For the *Pingpong* and *Stencil* benchmarks we have seen an increase over 200%. For message intensive benchmarks such as *Shift*, the execution time is over 600% longer compared to the *message target* logging. These large overheads are due to the large amount of data logged when the size of the messages is considered. The message size distribution varies depending on the benchmark. Some benchmarks such as *Pingpong* are heterogeneous in their message sizes, as shown in Figure 4. Benchmarks *Stencil* and *Shift* have fixed sized messages of 64 bytes and 128 bytes, respectively.

The rest of the experiments use the *message target* logging, which logs the performance counters, number of messages, and message sender/receiver information at a low overhead.

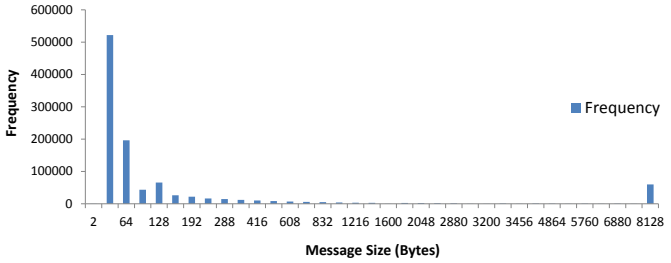


Fig. 4. *Pingpong* message size histogram. The majority of *Pingpong* messages are small; however, there are also a significant number large messages. As the broadcast benchmark is derived from *Pingpong* it has the same distribution of message sizes.

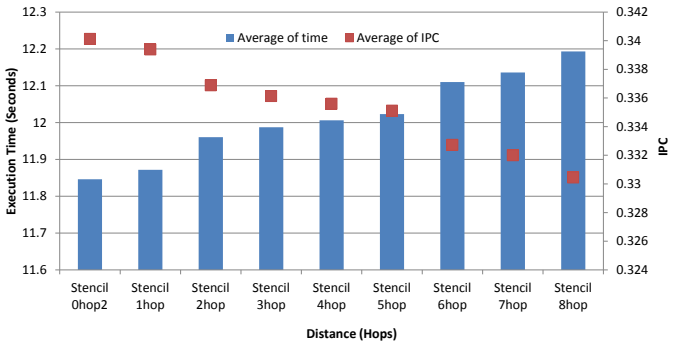


Fig. 5. *Stencil* with one pair of cores. The distance between the cores increases from local communication on the same router to the maximum distance spanning 8 routers.

B. Impact of Communication Distance

Next, we analyze how the distance between communicating cores affects performance. We look at the case of a single pair of cores that are running on the SCC. Figure 5 demonstrates that as the distance between the cores increases, the execution time increases. In the figure, we plot the execution time of *Stencil* as the distance between cores is increased from 0 hops (local) to the maximum distance of 8 hops (cores 0-47). There are clear linear trends for both the IPC and the execution time. *Stencil* is chosen in this experiment as it demonstrates the largest difference in execution time owing to its high IPC (as outlined in Table I). Similar trends can be seen for *Shift*.

C. Impact of Memory Accesses

To measure the impact of memory accesses, we keep the distance constant but increase the number of cores (i.e., number of pairs simultaneously running). In this way, we expect to increase the accesses to the main memory. In this experiment, we do not see any measurable difference in execution time for *Stencil* or *Shift*, as their memory access intensity is low. The *Share* benchmark, which has a high memory accesses density at 0.05 (see Table I), is prone to significant delays when there is memory contention. Figure 6 demonstrates this point. We see significant delay when 24 pairs of cores are concurrently executing a benchmark that is heavy in memory accesses. The combined load of 24 pairs accessing memory is saturating the memory access bandwidth and causing the delay. While this effect is due to the uncached accesses to the DRAM and specific to the SCC, the trend is observed in many multicore applications which become memory bound.

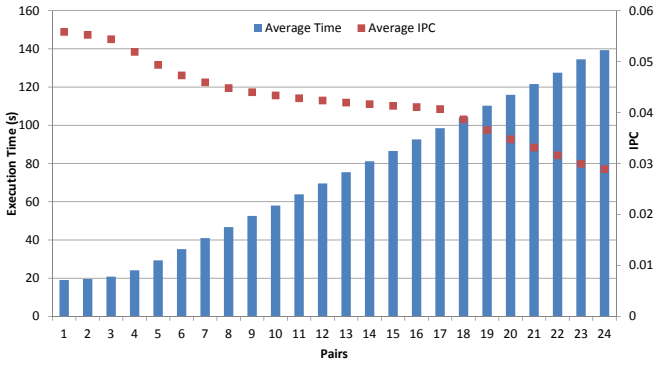


Fig. 6. Execution time of *Share* with local communication, as a function of the number of pairs executed concurrently.

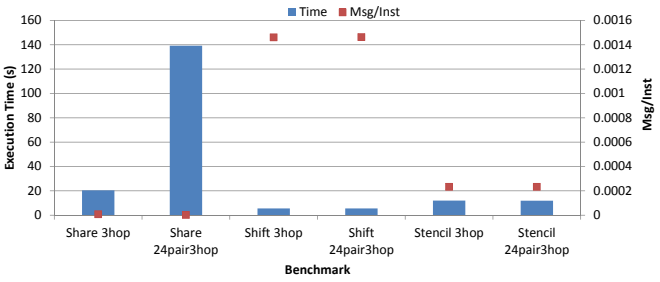


Fig. 7. *Share*, *Shift*, and *Stencil* with 3 hops communication. One pair compared to 24 pairs executed concurrently.

D. Impact of Network Contention

For exploring the impact of network contention, we compare 3-hop communication of a single pair of a benchmark versus running 24 pairs. *Shift*, which has low memory accesses but high message density, does not exhibit significant changes in execution time when comparing one pair with 24 concurrent pairs; this is visualized in Figure 7. When we look at *Stencil*, which has both memory accesses and messages, we still do not see significant differences in execution time as seen in Figure 7. In fact, the only major difference occurs for *Share*, owing to its memory access intensity. We believe these benchmarks have not been able to cause network contention; therefore, the dominant effect on the execution time is the memory access frequency in the figure.

E. Impact of Broadcast Messages

Next, we analyze performance for applications that heavily utilize broadcast messages. We run our *Bcast* benchmark for this experiment. The benchmark is an adaptation of the *Pingpong* benchmark, so as in *Pingpong*, *Bcast* sends many messages of different sizes. Instead of sending the messages to a specific core, *Bcast* sends messages to all of the receiver cores in a one to N broadcast system. Figure 8 demonstrates that as the number of cores in the broadcast increases we have significantly slower execution. It is particularly interesting that there is a peak IPC at $N = 8$ cores. This peak suggests that when $N > 8$ for the *Bcast* benchmark, the performance of the sender core and the network become bottlenecks.

Figure 9 demonstrates how as the number of cores in the broadcast increases, the messages per instruction increases

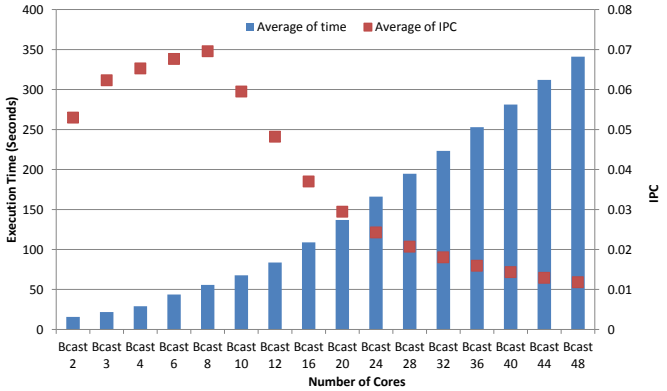


Fig. 8. Execution of Bcast with respect to number of cores. As the number of cores increase we see the growth in execution time.

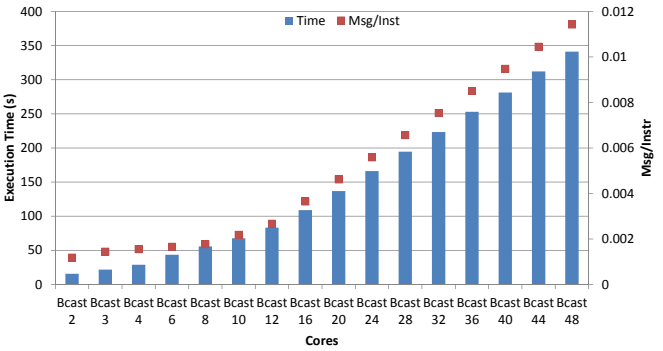


Fig. 9. Broadcast with increasing number of cores. As the number of cores increase we see a higher number of messages per instruction.

with it. Again we see that at $N = 8$ cores there is a local inflection point. This helps confirm that for this particular broadcast benchmark, broadcasting to a large number of cores saturates the traffic from the sender core, which in turn causes delays. This result highlights the importance of carefully optimizing broadcasting to ensure desirable performance levels. A potential optimization policy would be to allow for broadcasting to a small number of cores at a given time interval.

F. Power and Energy Evaluation

As part of our analysis, we also investigate the power and energy consumption for each benchmark. Figure 10 compares the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks in 24-pair 0-hop (local communication) configuration. We see that at full utilization of all 48 cores, a significant difference exists in the amount of power drawn by each benchmark. The *Share* benchmark, heavy in memory accesses and low in messages (see Table I), has relatively low power consumption compared to the *Shift* and *Stencil* benchmarks which have significantly higher IPC and power consumption. Overall, IPC is a reasonable indicator of the power consumption level.

Looking at power alone is often not sufficient to make an assessment of energy efficiency. Figure 11 compares energy-delay product (EDP) (delay normalized to 100 M instructions) for *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks in 24-pair 0-hop configuration. Again, significant differences exist in the EDP across the benchmarks. The high EDP in *Share* is a result

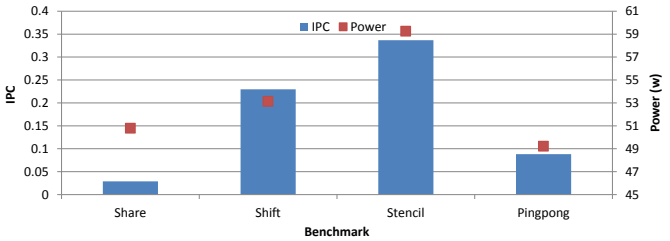


Fig. 10. Comparing IPC vs power for the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks. All benchmarks were executed with 24 pairs of cores, all with local communication.

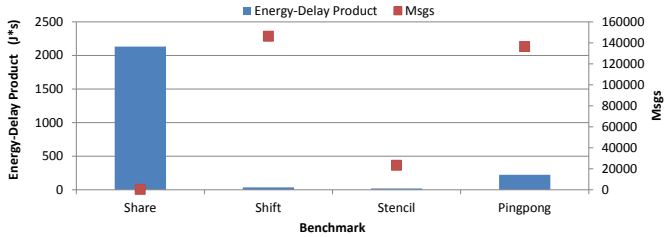


Fig. 11. Comparing EDP vs. number of messages for the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks. All benchmarks were executed with 24 pairs of cores, all with local communication.

of the high memory intensity and low IPC, which cause high delay. *Stencil* has the highest IPC, a low number of messages, and a medium level of memory accesses, which jointly explain the low EDP. *Shift* and *Pingpong* both have a considerable amount of messages. However, *Pingpong* misses a lot in the L1 cache, resulting in lower performance. Thus, its EDP is higher compared to *Shift*.

We have also compared running one pair of *Share* against 24 pairs. For one pair the power consumed is 31.616 Watts. When 24 pairs are run concurrently the power consumed jumps to 50.768 Watts. The power drawn follows linearly with the number of active cores. Due to the offset and leakage power consumption of the chip, running the system with a large number of active cores when possible is significantly more energy-efficient (up to 4X reduction in EDP per core among the benchmark set).

V. Related Work

There has been several projects relevant to the design, development, and experimental exploration of the Intel SCC. As part of Intel's Tera-scale project, the Polaris 80-core chip can be regarded as the predecessor of the SCC [8]. The main purpose of the Polaris chip was to explore manycore architectures that use on-die mesh networks for communication. However, unlike the SCC, it only supported a very small instruction set and lacked corresponding software packages that facilitate manycore application research [10].

Previous work describes low-level details of the SCC processor hardware [4]. Special focus is given to topics regarding L2 cache policies and the routing scheme of the mesh network. Other recent research on the SCC looks at benchmark performance in RCCE focusing on the effects of message sizes [7]. The authors also provide detailed performance analysis of message buffer availability in RCCE [7].

Another related area is the development of the message passing support. The RCCE API is kept small and does not implement all of the features of MPI. For example, RCCE only provides blocking (synchronous) send and receive functions, whereas the MPI standard also defines non-blocking communication functions. For this reason, some researchers have started to extend RCCE with new communication capabilities, such as the ability to pass messages asynchronously [3].

VI. Conclusion

Future manycore systems are expected to include on-chip networks instead of the shared buses in current multicore chips. MPI is one of the promising candidates to manage the inter-core communication over the network on manycore systems. This paper investigated the performance and power impact of the message traffic on the SCC. We have first described the monitoring infrastructure and the SW applications we have developed for the experimental exploration. Using our low-overhead monitoring infrastructure, we have demonstrated results on the effects of the message traffic, core performance characteristics, and memory access frequency on the system performance. We have also contrasted the benchmarks based on their power profiles and their energy delay product. Overall, the paper provides valuable tools and insights to researchers in the manycore systems research area. For future work, we plan to analyze the traffic patterns in more detail, create various local and global network contention scenarios, investigate opportunities to track other performance metrics (such as L2 cache misses), and utilize the experimental results for designing energy-efficient workload management policies.

Acknowledgments

The authors thank the Intel Many-Core Applications Research Community. John-Nicholas Furst has been funded by the Undergraduate Research Opportunities Program at Boston University.

References

[1] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, March 1994.

[2] A. Bartolini, M. Sadri, J. N. Furst, A. K. Coskun, and L. Benini. Quantifying the impact of frequency scaling on the energy efficiency of the single-chip cloud computer. In *Design, Automation, and Test in Europe (DATE)*, 2012.

[3] 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)*, pages 525 –532, July 2011.

[4] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108 –109, Feb. 2010.

[5] Intel. SCC external architecture specication (EAS). http://techresearch.intel.com/spaw2/uploads/files//SCC_EAS.pdf.

[6] M. A. Khan, C. Hankendi, A. K. Coskun, and M. C. Herbordt. Software optimization for performance, energy, and thermal distribution: Initial case studies. In *IEEE International Workshop on Thermal Modeling and Management: From Chips to Data Centers (TEMM), IGCC*, 2012.

[7] T. G. Mattson et al. The 48-core SCC processor: the programmer's view. In *High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –11, Nov. 2010.

[8] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –11, Nov. 2008.

[9] T. G. Mattson and R. F. van der Wijngaart. RCCE: a small library for many-core communication. *Intel Corporation*.

[10] S. R. Vangal et al. An 80-tile sub-100-W teraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29 –41, jan. 2008.