Analyzing Application Performance Bottlenecks on Intel’s SCC

Xu Liu*, John Mellor-Crummey*, and Nathan R. Tallent†
* Department of Computer Science, Rice University
Email: {xl10, johnmc}@rice.edu
† Pacific Northwest National Laboratory
Email: tallent@pnnl.gov

Abstract—Intel’s Single-Chip Cloud Computer (SCC) is an experimental multi-core processor designed to scale to hundreds of cores. However, compared to traditional multicore systems, the SCC places a high burden on application programmers. To date, SCC-based systems have had no performance tool that can help programmers rapidly analyze an application’s performance bottlenecks. To remedy this problem, we ported Rice University’s HPCTOOLKIT performance tools to the SCC in order to analyze hot loops with hardware performance counters, pinpoint scaling bottlenecks in context, identify load imbalance with call path traces, and identify time-varying performance problems such as serialization. This paper describes our experiences using HPCTOOLKIT to analyze the performance of the triangular-solve-with-multiple-right-hand-sides (Trsm) implementation from the Elemental library and the SCC-version of the NAS LU benchmark. HPCTOOLKIT enabled us to understand Trsm’s performance in detail and provided insight that enabled us to improve the performance of the NAS LU benchmark by more than 30%.

Index Terms—Intel SCC, HPCTOOLKIT, performance tools

I. INTRODUCTION

Intel’s Single-Chip Cloud Computer (SCC) [1] is an experimental multi-core processor designed to scale to hundreds of cores. The SCC integrates 48 P54C (x86) processor cores into an on-die 2D-mesh network. The cores are arranged into 24 tiles. The frequency of each tile can be configured individually. Instead of providing coherent caches to support a shared-memory programming model, the SCC provides unique message passing buffer (MPB) hardware to support a message passing programming model. Programming the SCC is like programming a cluster: each core accesses its own private memory and must explicitly manage interactions with other cores.

To make effective use of microprocessors such as the SCC, developers need tools that help them understand the interplay between SCC hardware and their application codes. Prior to this work, there was no performance tool available on the SCC to support analysis of application performance bottlenecks both within and across the tiles on the SCC. To address this problem, we ported Rice University’s HPCTOOLKIT performance tools to the SCC. HPCTOOLKIT is the first SCC performance tool that can (a) pinpoint hot spots and scaling bottlenecks in their full calling context using time or hardware performance counter metrics and (b) identify load imbalance and serialization using call path traces — all for less than 5% measurement overhead.

Section II of this paper briefly summarizes HPCTOOLKIT’s capabilities and describes issues that arose porting it to the SCC. Section III describes some case studies where we use HPCTOOLKIT to analyze codes on the SCC. Section IV concludes and identifies future work. Finally, Section V describes the current status of HPCTOOLKIT on the SCC.

II. HPCTOOLKIT ON THE SCC

HPCTOOLKIT is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multi-core systems to supercomputers. HPCTOOLKIT incorporates several novel measurement, attribution, analysis and presentation techniques that distinguish it from gprof [2], the only performance tool currently available on the SCC. gprof was designed to use an interval timer to trigger collection of a call graph profile on an individual processor core. In contrast, HPCTOOLKIT can (a) attribute wall clock time, hardware performance counter measurements, and derived metrics to their static and dynamic calling contexts [3], (b) identify scaling losses between executions at different scales [4], (c) quantify load imbalance [5], and (d) collect call path traces to show how an execution of a parallel program unfolds over time [6]. By using sampling instead of instrumentation, HPCTOOLKIT avoids blind spots and collects performance data with low overhead. Moreover, it collects performance data for fully optimized application binaries without the need to recompile.

The workflow of HPCTOOLKIT’s tools is shown in Figure 1. hpcrun is a call path profiler and tracer that preloads
a monitoring library into an application’s address space during application launch. This library configures sampling sources, which typically include an interval timer and hardware performance counters, with predefined sampling periods for generating asynchronous signals. On each signal, hpcrun captures the application’s execution context using stack unwinding and associates measured costs with the current context. To minimize memory overhead while monitoring long executions, hpcrun stores samples using a compact representation known as a calling context tree (CCT) [7].

To attribute performance metrics to loops and inlined code without perturbing measurements in any way, HPCTOOLKIT uses its hpcrect tool to recover an application’s program structure off line. hpcrect identifies source-level procedures, loops and inlining by analyzing an application’s object code, reconstructing its control flow graph, performing interval analysis (loop nest discovery), and making inferences based on information from an executable’s line map.

HPCTOOLKIT’s hpcprof combines execution measurement data gathered by hpcrun with static program structure from hpcrect to attribute performance metrics to source code and calling contexts. It integrates data associated with the same contexts from different threads and processes, and computes statistics such as the sum, average, standard deviation, min and max to summarize their range of metric values. To compare the performance of different executions, hpcprof can combine multiple measurement databases.

HPCTOOLKIT’s hpcviewer and hpctraceviewer are tools for interactively presenting profile and tracing data, respectively. The tools’ graphical interfaces have been designed to help an analyst rapidly diagnose performance problems. We demonstrate the utility of these interfaces in Section III.

Although most of the work to port HPCTOOLKIT to the SCC was straightforward, we encountered some challenges. First, Intel’s experimental SCC systems are heterogeneous systems composed of a Management Console PC (MCPC) host and a SCC chip. MCPC hardware is typically based on the x86-64 architecture while SCC cores are x86-based. hpcrun’s measurement library must be cross-compiled to execute on the SCC chip, while analysis tools need to be compiled for execution on the front end.

Second, each SCC processor core has limited private memory: only about 300 MB (with a maximum of 1.3 GB) and non-virtualized in our SCC platform. This limited memory posed a problem for an hpcrun utility that relies on a third-party library. To unwind call stacks for fully optimized code, hpcrun uses binary analysis to compute unwind recipes on the fly. As a part of that analysis, it is necessary to recover function boundaries in partially stripped code. To populate an initial set of function bounds, hpcrun uses the SymtabAPI library [8] to read an application binary’s symbol table. We found that on some applications the memory required by this library was too large to concurrently coexist with the application’s code and data. To solve this problem on the SCC, we precompute this information off-line on the MCPC as an application is launched and then use it while collecting measurements.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (mesh, processor)</td>
<td>800 MHz, 333 MHz</td>
</tr>
<tr>
<td>Compiler</td>
<td>icpc 8.1</td>
</tr>
<tr>
<td>MKL</td>
<td>MKL 8.1.1</td>
</tr>
<tr>
<td>Compilation options</td>
<td>-O3 -g</td>
</tr>
<tr>
<td>Work size</td>
<td>10K x 10K matrix</td>
</tr>
<tr>
<td>Sampling period</td>
<td>5 ms</td>
</tr>
</tbody>
</table>

Third, unlike typical Intel processors, the SCC’s P54C cores do not support sampling based on hardware counter events. (These P54C cores cannot generate overflow interrupts.) To collect measurements using hardware performance counters, we use the Linux interval timer as a sampling proxy. Each SCC process configures an interval timer to generate interrupts at a specified frequency. Two consecutive interrupts form an interval. At the end of each interval, we read P54C hardware counters, compute the number of events that occurred in this interval by subtracting the hardware counter values obtained at the start of the interval from the current counter values, and attribute the differences to the calling context at the current sample point. For all but the most rapidly varying calling contexts, the resulting profiles accurately reflect the distribution of events triggered throughout the program execution.

III. CASE STUDY

The NAS Parallel Benchmarks (NPB) and the High Performance Linpack (HPL) benchmark are supplied with the RCCE communication library. The Elemental benchmark was developed for the SCC as part of the FLAME project [9]. We use HPCTOOLKIT to study the performance of Elemental because it is the most sophisticated of the codes. We also use HPCTOOLKIT to study the NPB LU benchmark because it is used by Intel group to study the SCC performance [1].

Elemental is a scalable version of BLAS and LAPACK coded in C++. It hides the implementation of matrices and vectors with objects to improve programmer productivity and code maintainability. In Elemental, all communication between processes is collective. The authors of Elemental wrote their own collective communication because the RCCE library does not provide collectives. Elemental uses Intel’s MKL math library for its computational kernels. Although we do not have the source code of MKL, HPCTOOLKIT can still evaluate its performance. Of the more than 20 Elemental benchmarks, we chose to study Trsm, which performs a triangular solve with multiple right-hand sides. The SCC configuration and parameters for Trsm for our study are described in Table I. We will use HPCTOOLKIT to analyze Trsm’s hot spots, scaling gains and losses, and load imbalance.

The NAS LU benchmark “uses a ‘pencil decomposition’ to assign a column block of a 3D discretization grid to each core. A 2D pipeline algorithm is used to propagate a wavefront communication pattern across the cores” [1]. For our study of LU, we used input class A, 32 cores, and compiled with Intel’s icc 8.1 with -03 -g options. We sampled LU’s execution every 5 milliseconds on each core. LU suffers performance degradation from serialization. We pinpointed problematic
code sections using `hpctraceviewer`; fixing them boosted performance by 33.7%.

### A. Analyzing Hot Spots and Loops with Performance Counters

We identify Trsm’s hot spots at the function and loop levels when executed on 48 cores. Figure 2 shows the calling context view in `hpctraceviewer`, which consists of three panes. The top pane displays the source code; the bottom-left pane shows a navigation pane that, for the calling context view, shows call paths; and the bottom-right pane shows performance metrics. For this experiment, our metrics include time and two hardware events—L1 cache accesses and misses. Expanding the call path from `main`, we find that `DGEMM` referenced in a loop in file `TrsmLLN.cpp` is the hot spot. It consumes 60.7% of the total execution time. The MKL `DGEMM` makes good use of the cache. It performs 86% L1 cache accesses in the execution with a 3% miss rate. Although it performs 86% of the cache accesses, it is responsible for only 42% of the cache misses. Drilling down the call path from `DGEMM`, we can see the hot loops in the MKL library. Called in the innermost loop, `mkl_blas_def_dinner` is the most time-consuming routine in this call path, which takes 30.2% of total execution time. This routine has good locality, with 45.4% of the total L1 cache accesses responsible for only 24.5% of the total L1 cache misses. To analyze the performance of a routine independent of its calling context, we switch to `hpctraceviewer`’s Flat view, shown in Figure 3. This view presents the cost of each routine accumulated from all of the routine’s calling contexts. The figure shows that the total time consumed in `mkl_blas_def_dinner` is 48.8% of total execution time, while the cache accesses and misses caused by this function are 72.5% and 33.9% of total accesses and misses, respectively.

### B. Pinpointing Scaling Bottlenecks in Their Calling Context

HPCTOOLKIT can compute a context-sensitive scaling loss metric using measurements collected from two or more program executions. A strong scaling experiment involves solving the same problem on different numbers of processors. Consider measuring the performance on `a` processes and again on `b` processes, where `b > a`. The formula for computing the percent scaling loss for any context `c` in the two executions is as follows:

Fig. 2. Calling context view of Trsm’s performance data showing the functions and loops along the hot call path.

Fig. 3. Flat view of Trsm’s performance data showing hot functions according to the static program structure.

Fig. 4. Flat view of Trsm’s performance data showing scaling loss at the function level.
Figure 5 shows a strong scaling analysis for Trsm running on 24 cores and 48 cores in a flat view. The overall scalability of Trsm is good with only 5.72% scaling loss. Trsm demonstrates excellent scalability in the computational routine, DGEMM. There is a very slight (0.121%) super-linear speedup, possibly because the on-chip cache used doubles as the computation is scaled from 24 cores to 48 cores. We can collect the hardware event \texttt{WAITING\_FOR\_DATA\_MEMORY\_READ\_STALL\_DURATION} to verify our intuition. This hardware event accumulates processor pipeline stall cycles due to waiting for a data fetch from memory. DGEMM running on 48 cores costs 0.6% less time waiting for the memory than running on 24 cores. Though 0.121% and 0.6% may be so small as to be affected by the statistical error, both measurements show excellent performance in Trsm’s computational kernel.

In contrast, data movement in Trsm does not scale as well as computation. The collective communication routine \texttt{RCCE\_allgather} and \texttt{memcpy} show 19.0% and 12.6% scaling losses respectively. Unsurprisingly, the \texttt{RCCE\_allgather} collective has scalability losses as the number of processes doubles from 24 to 48 since one or more additional rounds of communication will be required. \texttt{memcpy} is used by RCCE to copy data from remote MPBs to the local memory and its scaling loss can be attributed to a 31.5% increase in data transferred by Trsm when scaling from 24 to 48 cores.

Overall, Trsm is a computationally intensive application. Because its computation consumes much more time than its communication, the total scaling loss of the whole benchmark is less than 6%.

C. Understanding Load Imbalance and Temporal Patterns

Figure 5 shows a call path trace of a 48-core Trsm execution in hpctraceviewer. The trace view (top left) is organized as a collection of time lines. Time advances along the X axis from left to right. Processes are ordered by ranks along the Y axis. Each color along a time line represents a procedure active when a process was sampled. The trace view can display the activity of the processes at multiple levels of abstraction by displaying them at different call stack depths. In this figure, the trace view shows process activity at depth four. Insets within the top-left pane show expanded views of highlighted portions of 1.5 second intervals near the beginning and the end of the execution. The bottom-left pane shows call stack depths vs. time for the process marked by the cursor. The top-right pane controls the depth of call stack shown in the trace data.
Fig. 6. Space-time view of NAS LU execution. Processes are arranged top to bottom along the Y-axis. Time proceeds left to right along the X-axis. The skewed pattern indicates that the computation is serialized in some way. A process can’t complete a send until its predecessor completes its own send. This bottleneck occurs in the exchange_3 communication routine.

From Figure 5, we can see the execution consists of multiple iterations. In each iteration, computation (shown in green) and communication (blue lines between green blocks) can be clearly distinguished. From this high-level view, Trsm has excellent load balance because all processes complete the computation in each iteration at nearly the same time. The insets show that there is more computation (dark green) in early iterations than in later ones. The differences between the process time lines shown in the insets indicate slight load imbalance. By comparing profiles for each of the processors using data in columns not shown in Figure 3, we identified a 4.2% computation imbalance in Gemm; similarly, the data movement routines memcpy and allgather have a 10.0% imbalance.

D. Identify code sections with serialization

Serialization reduces parallelism and degrades performance. We now combine HPCTOOLKIT’s profiling and tracing data to highlight serialization. We use hpctraceviewer to analyze an execution’s behavior over time and pinpoint serialization. We use hpcviewer's profiles to estimate the magnitude of serialization and the opportunity for optimization.

Similar to Trsm, LU performs computation and communication in an interleaved fashion. Figure 6 shows communication due to data exchanges along coordinate dimensions within one iteration of LU’s execution. The red rectangle highlights the communication pattern among 32 processes corresponding to the data exchange along one coordinate dimension. The 32 processes execute as four groups of eight. Communication for one group of eight is highlighted by the yellow rectangle. The annotations of the trace view show the pattern of sends and receives resulting from execution of exchange_3. The staggered completion of receives indicates serialization. During an instance of exchange_3, a process cannot complete a send until its successor has issued a receive. A process won’t receive from its predecessor until it has completed its own send. This serializes the data exchange. This happens with both forward and reverse communication along a partitioned data dimension. Analysis using hpcviewer indicates that exchange_3 consumes 49.3% of the total execution time so it is worthy of investigation for improvement.

To avoid serialization of communication in exchange_3, we changed the algorithm so that odd processes send first and receive second; even processes receive first and send second. That way, each sender/receiver pair may exchange data concurrently. This strategy is similar to the approach used to avoid deadlock for communication in a ring [10]. Figure 7 shows a trace view of the optimized communication for one process group. In this figure, deep green represents receive and light green represents send. Although the processes to not begin the communication phase at exactly the same time, there is no serialization within the phase itself—a clear improvement over the original communication pattern shown in Figure 6. The optimization reduces execution time of exchange_3 by 71.2% and the whole LU program by 33.7%.
Fig. 7. A space-time view of the optimized exchange_3 communication in NAS LU that caused the serialization shown in Figure 6. The absence of a skewed pattern here shows that the optimization avoids the serialization in the original implementation.

IV. CONCLUSIONS

Because scalable parallel performance is critical on the SCC, developers need powerful tools to pinpoint and understand performance bottlenecks. We have shown that HPCTOOLKIT’s analyses can provide unique insight on SCC applications. Moreover, HPCTOOLKIT has low time and memory overhead. For the applications we analyzed, when using a reasonable sampling rate, the measurement overhead was at most 5% of total execution time and it required less than 10MB of memory per core.

For future work, our plans are two-fold. First, we will leverage HPCTOOLKIT and available benchmarks to evaluate the network on the SCC and understand how network congestion affects performance. Second, we plan to monitor whole chip power consumption and thermal sensor values on each SCC tile and correlate energy consumption and heat to calling contexts over time. Our interest is in determining whether such information might be helpful for fine-grained power management on the SCC.

V. STATUS

Source code for the SCC version of the HPCTOOLKIT performance tools is available at [http://outreach.scidac.gov/svn/hpctoolkit/branches/scc](http://outreach.scidac.gov/svn/hpctoolkit/branches/scc). Currently, HPCTOOLKIT works well on the SCC with 1.3.0 and 1.4.x (1.4.1 and 1.4.2) sccKits, which provide 2.6.18 and 2.6.38 SCC Linux kernels respectively.

ACKNOWLEDGMENTS

We thank Mike Fagan and Mark Krentel for their help porting HPCTOOLKIT to the SCC. We especially thank Bowden Kelly for his work creating a 32-bit version of hpcrun’s call stack unwinder; without his efforts, collecting call path profiles on the SCC would not have been possible.

REFERENCES


