An Efficient Front-End for Timing-Directed Parallel Simulation of Multi-Core System

Zhenjiang Dong, Jun Wang, George Riley, Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
{zdong30, jun.wang, riley, sudha}@ece.gatech.edu

ABSTRACT

Manifold is a parallel simulation framework for multi-core systems. For full-system simulation, Manifold adopts the timing-directed simulation paradigm that separates the simulation into a functional front-end and a timing back-end. Components in the front-end perform functional simulation of the cores and send streams of instructions to the back-end to simulate the timing behavior. In its current design, Manifold uses the QSim multi-core emulator as the front-end, which communicates with the back-end through network sockets. Experiments have shown that the latency of the socket communications has a significant impact on the overall simulation performance. This paper presents a novel method that attempts to hide the TCP/IP latency for the back-end by creating proxy processes as an intermediary between the front-end and the back-end. The proxies serve as clients to the QSim server in the front-end, and as servers to the back-end. They interact with the QSim server through sockets, while working with the back-end in a producer-consumer manner using shared memory segments. Experiments show that this method can completely hide the TCP/IP latency for the back-end. The back-end can always get its instructions from the shared memory without waiting for the QSim server. The overhead of getting inputs for the back-end simulation is reduced to almost zero. As confirmed by our experiments, this improvement causes some side effects that together lead to significant improvements in overall simulation performance. In testing of system models with up to 64 cores, we have achieved from 29% to 51% improvement in simulation performance.

Categories and Subject Descriptors

C.0 [**Computer System Organization**]: Modeling of computer architecture

General Terms

Experimentation, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Simutools 2014, March 17-19, Lisbon, Portugal
Copyright © 2014 ICST 978-1-63190-007-5
DOI 10.4108/icst.simutools.2014.254638

Keywords

parallel simulation, full-system model, timing-directed, multi-core

1. INTRODUCTION

Simulation is an important tool used by researchers and industry CPU architects for evaluating and analyzing future architecture designs. In one important class of mirco-architecture simulations, which we shall refer to as *timing-directed simulation*, the simulation system is divided into two separate but complementary parts: *functional simulation* and *timing simulation*. The functional simulation, also known as the *front-end*, emulates the behavior of the target system. For example, emulators such as SimOS [12], Qsim [8] and AMD's SimNow^{TM} [2] are used for functional simulation. Functional simulation is very precise in terms of functional behavior and fast enough to run at the speed close to that of native execution. However, functional simulations focus on the correctness of functional behavior and typically don't have accurate timing for devices in the system. On the other hand, timing simulation, or the *back-end*, is built with the simulation models of architecture components and used to evaluate the performance of target system. Timing simulation is one or more order of magnitude slower than the functional simulation, but it models the operation latency of architecture components.

In timing-directed simulation for micro-architectures, functional simulation is responsible for generating correct instruction flow or events stream for architecture components in timing simulation. Timing simulation uses the instructions or events to drive architecture components, and sends feedback to functional simulation based on the state of architecture components. Then, the functional simulation adjusts its own state according to the feedback. The interactive features of timing-directed simulation make it ideal for simulation of complex system that can change functional behavior dynamically during run time such as multi-core system. Also due to the interactive nature, the communication method between the two parts is important for overall performance of simulation.

Manifold [14] is a software framework for building *parallel* simulation of multi-core systems following the timing-directed simulation paradigm. In a Manifold full-system simulation, the front-end is the QSim [8] multi-core emulator, and the back-end consists of timing models from Manifold's repository. QSim boots a Linux OS and executes

multi-threaded applications on emulated cores. Through a multi-threaded server, called QSim Server, the back-end core models can move the emulation forward and receive the executed instructions as a result. The back-end uses the instruction flows as input to simulate the timing.

In the current design, the back-end core model and the QSim Server use sockets to communicate. A core model issues a request and blocks until it receives the requested instructions. We have found the round-trip delay of the TCP/IP packets to be a detrimental factor to better performance, and it is desirable to hide this latency.

In this paper we propose an innovative method to hide the TCP/IP latency found in the current client-server design. We introduce proxy processes as an intermediary between the QSim server and the back-end. The proxy acts as client to the server and as server to the back-end. It gets instructions from the server over sockets and puts the results in shared-memory segments. The back-end, instead of interacting with the server directly, obtains instructions from the shared-memory segments. Experiments have shown that this method successfully hides the TCP/IP latency, makes the core models acquire inputs more efficiently, and as a result, significantly improves the overall simulation performance.

In the rest of the paper, we first introduce background knowledge and related work in Section 2. In Section 3 we presents the design and implementation of the proxy process. Experimental results and our analysis are shown in Section 4, and finally Section 5 contains our conclusions.

2. BACKGROUND AND RELATED WORK

With the complexity of multi-core design growing sharply in recent years, a few parallel simulation systems have emerged, including [14], [10], [11] and [1]. Our work is built upon the Manifold Project [14], an open source software project that provides a scalable infrastructure for modeling and simulation of many-core architectures.

As far as timing-directed simulation is concerned, the most similar to Manifold is COTSon [1]. COTSon also separates simulation into functional simulation and timing simulation. It uses AMD's proprietary SimNow ^{TM} [2] as the functional simulator, which operates at each node of a cluster in a distributed manner and produces a stream of events for the respective CPU timing model. Graphite does not follow the timing-directed simulation paradigm. It uses PIN [6] traces instead. The PIN execution is instrumented to trap events that drive the timing models. The Structural Simulation Toolkit (SST) [11] is similar to Manifold. SST also supports QSim. It, however, does not use the Qsim server as front-end. Therefore, it does not have the same issue as Manifold, namely, to get instructions efficiently from the QSim server to the back-end.

Manifold supports both trace-driven and timing-directed simulations. In trace-driven simulation Manifold achieved reasonable speedup compared to sequential simulation and demonstrated good scalability in tests of models with up to 128 cores [4]. However, little work has been done to study and improve the performance of the timing-directed simulation

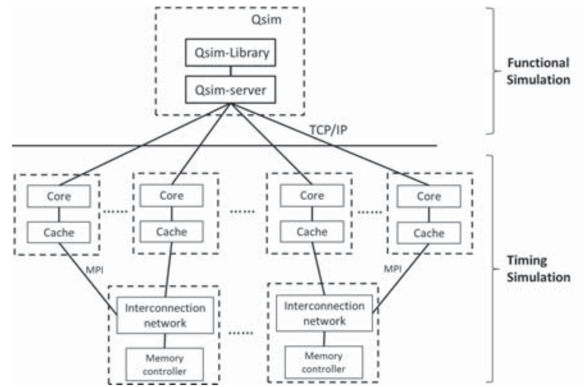

Figure 1: Manifold's timing-directed simulation model with client-server design.

of Manifold, and this is the purpose of this paper.

The timing-directed simulation of Manifold can be decomposed into two major parts – the functional simulation front-end and the timing simulation back-end, as shown in Figure 1. In this figure, we can see that the back-end system model is composed of a certain number of processor nodes and memory controllers that are connected to an interconnection network. Each processor node consists of a core and one or more levels of cache. For parallel simulation, components are assigned to different processes known as logical processes (LPs).

The front-end is a QEMU-based [7] thread safe multi-core emulation library called Qsim [8]. QSim can boot a Linux OS and emulates the execution of a multi-threaded application on a number of virtual cores. The back-end core models send requests to QSim, which in turn emulates the execution of instructions on corresponding virtual cores. The executed instructions and related information such as virtual and physical addresses of the instructions are then sent back to the back-end models. Instruction execution progress of each virtual core is controlled by the corresponding back-end core model.

For full-system simulation, QSim provides a multi-threaded server to handle requests from the back-end. As shown in Figure 1, the back-end models and the server communicate over TCP/IP using sockets. When a core model needs instructions, it sends a request to the server and then blocks for the response. Once the instructions are received, it resumes its simulation activities. In other words, whenever a core model in the back-end needs instructions, TCP/IP communications are incurred and the core model cannot progress until the TCP/IP transaction completes. Since we cannot eliminate the TCP/IP latency, it is highly desirable to devise a method that hides the latency as much as possible from the back-end. We expect such a scheme would make the back-end core models run more efficiently and improve the overall simulation performance.

3. DESIGN OF QSIM PROXY

The client-server based design, which we introduced above, is shown in Figure 1. In the following we shall refer to this as the *baseline* design. In the baseline design, the back-

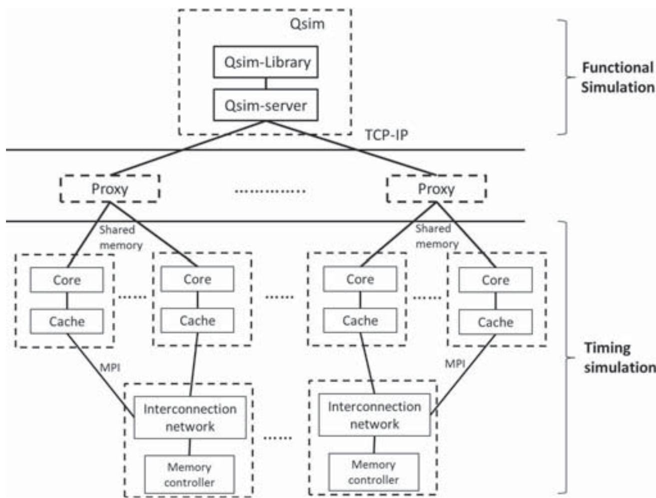

Figure 2: Manifold parallel simulation with Proxy.

end core models communicate directly with the QSim Server using sockets. The multi-threaded server spawns one thread for each core model. Whenever a core model runs out of instructions, it starts a transaction with the server to get instructions. The core model would block until it receives the server's reply. As stated above, it is the TCP/IP latency that we wish to hide from the back-end in order to improve performance.

We attempt to achieve this by creating proxy processes between the server and the back-end, as shown in Figure 2. Compared with the baseline design, there are two major differences:

1. In this design we introduce to the system a number of proxy processes. The proxies, together with the server, form a new, improved front-end.
2. The back-end core models are modified. Instead of interacting with the server directly, the core models would get instructions from the proxy.

The proxy acts as an intermediary between the server and the back-end. It serves as a client to the server and as a server to the back-end. To get instructions from the server to the back-end, the proxy communicates with the server using sockets, just as the back-end does in the baseline design. Once instructions are received, they are first stored in an internal buffer and then copied to shared memory segments for the consumption by the core models. We choose shared memory segments because they are a highly efficient means for inter-process communication. Each proxy is a multi-threaded process, with one thread for each core model it serves. A proxy serves all the core models running on the same host machine.

Each proxy thread and the core model it serves form a producer-consumer relationship. The proxy thread puts instructions in the shared memory segment, and the core model removes them. This is shown in Figure 3. To simplify inter-process synchronization, a circular buffer is implemented on

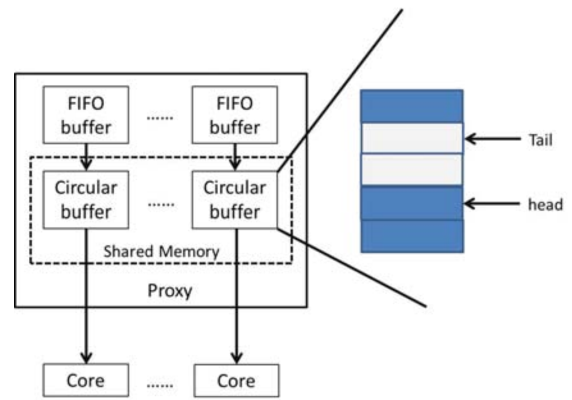

Figure 3: Implementation of Proxy.

the shared memory segment. Two additional pointers are allocated, for the head and the tail of the circular buffer. The producer (proxy thread) only modifies the tail and the consumer (core model) only modifies the head. Therefore, no synchronization mechanism, such as semaphore, is required. To prevent buffer overflow, the thread uses an FIFO buffer to hold instructions it receives from the server. Instructions are eventually copied from the FIFO buffer to the shared memory segment (circular buffer).

The shared memory segments serve as a kind of "pre-fetch buffer". The back-end would get instructions from this buffer with almost no overhead, instead of through the high-latency TCP/IP link. In order to completely hide the TCP/IP latency for the core models, a proxy thread only need to ensure the circular buffer is not empty. To do this as much as possible, the thread keeps monitoring the contents of the circular buffer. As soon as it falls below a pre-defined threshold, the thread sends request to the server to get more instructions. The action of the proxy thread is given in Algorithm 1.

Algorithm 1 Proxy thread.

1: **while** true **do**
2:     **if** circular buffer size below threshold **then**
3:         get instructions from server into FIFO
4:     **end if**
5:     **while** FIFO not empty AND circular buffer not full **do**
6:         $Instruction \leftarrow$ remove first item from FIFO
7:         Write $Instruction$ to circular buffer
8:     **end while**
9:     sleep if did some work
10: **end while**

The shared memory segment is wrapped in a class which provides the following functions, among others:

- `write()`: this function is used by the proxy to write instructions to the shared memory.
- `read()`: this function is used by the back-end to read instructions from the shared memory.
- `is_empty()`: this function returns true if the shared memory segment is empty.
- `is_full()`: this function returns true if the shared memory segment is full.

4. EXPERIMENTAL RESULTS

We have implemented the two different methods for getting instructions from the front-end to the back-end. This section will compare the performance in terms of simulation time for each method, and present our analysis of the results.

4.1 Design of Experiments

We have built and tested system models for multi-core system with 16, 32, and 64 cores. For each system model, every core has a private level 1 cache and shared level 2 cache. All cores and their caches are connected to a single interconnection network, and one memory controller is created for every 8 cores. The interconnection networks we use is torus network, and 4×5, 6×6 and 9×8 torus networks are used in 16-, 32- and 64-core system models respectively. The core model in our tests is a cycle-level out-of-order x86 model called Zesto [9], and the cache models implement the Modified-Exclusive-Share-Invalid (MESI) coherence protocol [13]. A credit-based flow control protocol has been implemented along the core-cache-network path, and among routers inside the network. All the architecture components in the back-end are registered to the same clock and run at the same frequency. For parallel simulations, we use an enhanced null message algorithm called Forecast Null Message (FNM) [15].

We conduct our simulation on a Linux cluster that has 8 nodes with two Intel Xeon X5670 6-core CPUs on each node. The Intel Xeon X5670 6-core CPU has two logical threads on each core. Therefore there are 24 total hardware threads on each node. The operating system is RHEL release 6.3 with Open MPI 1.5.4. In all tests, the QSim server is assigned to its own node that runs no simulation processes, and we ensure each LP of the timing simulation has its own hardware thread. The back-end has two different types of LPs – core-cache LPs each assigned 2 Zesto cores and their caches, and network LPs each assigned a certain number of routers. In tests for the 16-core system model, 1 node is used to run the proxy and all the back-end. Simulations for 32-core system use 2 nodes, all 6 network LPs are assigned to one node with 6 core-cache LPs, and the rest of core-cache LPs are assigned to the other node. For the 64-core system, we use 3 nodes to run our simulation, one of them runs all 8 network LPs along with 8 core-cache LPs, while the rest of core-cache LPs are divided into two equal groups and each group is assigned to one of the two remaining nodes.

The size of the shared memory segments we used for all the tests is 262144 (256K) bytes, which is the best value we find during our preliminary tests. Tests for each system have been conducted against 6 randomly chosen benchmarks. The benchmarks we used are from two different benchmark sets, *vips*, *streamcluster* and *freqmine* are from the PARSEC [3], while *barnes*, *cholesky* and *fmm* are from the SPLASH-2 [16]. Tests are conducted for 50 million simulation cycles. For simplicity, we only examine the wall-clock simulation time of the core-cache LPs in this paper. The simulation time is further decomposed into time for getting instructions, safety check, processing events, sending null messages, and receiving incoming messages. The time for getting instructions is the total time it takes for a Zesto core in one LP to get instructions from Qsim server or proxy. The time for safety check is the time consumed by LPs to check if it is safe to process the next events, which is necessary in null-message-based parallel simulation. The time for processing events is the time for each LP to process local simulation events (time for getting instructions is part of this time). The time for sending null messages is the average time spent by LPs to send null messages. Finally, the time for receiving incoming messages is the time spent by an LP in processing incoming messages from other LPs. The total time is recorded with Linux *time* command, and the rest of time consumptions are recorded with the *clock_gettime()* [5] function. We insert two calls of the *clock_gettime()* function at the beginning and end of the sections that respectively perform safety check, event processing, sending null messages, and receiving incoming messages, and the execution time of each section is accumulated throughout the simulation.

4.2 Comparison of Designs and Analysis

The experiment results show clearly the advantage that the Proxy design has over the baseline design. Further, the latency of the communication with the front-end appears to have side effects on the performance of other tasks as listed above. The side effects lead to performance difference far beyond the effect of communication latency itself and hiding the communication latency is the major reason for the better performance of the Proxy design.

4.2.1 Comparison of Designs

Tables 1, 2, and 3 show six types of time consumption in seconds for each benchmark and the percentage of performance improvement of Proxy design over the baseline design in the parentheses. As shown in Tables 1, in tests for the 16-core system, the Proxy design has better performance than baseline design for all benchmarks. The simulation time is from 30% to 49% less compared to baseline design. The difference of event processing time between the two designs is relatively small compared to that of other time consumptions. However, the other five types of time consumptions show great difference when the front-end changes from Qsim Server to Proxy. The time for getting instructions in the baseline design is well above 15 times more than that of Proxy design, while Proxy design has 54% to 72% less time for sending null messages, 48% to 67% less time for safety check, and 49% to 67% less time for processing incoming messages.

Table 1: Simulation running time in seconds for 16-core model.

Proxy						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	9.5	607.0	1928.7	218.0	655.5	3409.2 (30%)
cholesky	14.1	515.2	2148.6	181.5	569.8	3415.1 (43%)
fmm	14.0	511.9	2134.3	177.3	565.9	3389.4 (49%)
freqmine	13.4	506.1	2063.6	176.6	560.0	3306.2 (44%)
streamcluster	13.8	513.3	2121.2	176.5	568.5	3379.5 (42%)
vips	12.8	533.9	2070.7	186.8	584.9	3376.4 (47%)
Baseline						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	145.9	1180.6	1930.2	475.3	1285.7	4871.8
cholesky	275.0	1426.8	2458.0	585.2	1571.9	6042.9
fmm	307.9	1588.2	2635.2	634.9	1730.6	6588.8
freqmine	272.0	1411.8	2393.0	557.3	1543.6	5905.7
streamcluster	266.5	1382.1	2390.2	574.4	1524.0	5870.7
vips	286.7	1515.4	2527.5	647.4	1690.4	6380.7

Table 2 presents the results for tests for the 32-core system.

The Proxy design consistently runs from 32% to 41% faster than baseline design. In comparison of the time for sending null messages, safety check, and processing incoming messages, the Proxy design spends from 40% to 53% less time in these three execution sections than the baseline design, which indicates the baseline design spends significantly more time in waiting without making actual progress of simulation. And, the event processing time for the baseline design is just about 15% more than that of the Proxy design. Apparently the useful execution time is roughly the same when leaving out the difference in time for getting instructions.

Table 2: Simulation running time in seconds for 32-core model.

Proxy						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	6.4	860.5	1429.7	367.6	965.5	3623.3 (41%)
cholesky	6.2	898.0	1374.0	377.6	1000.3	3649.9 (33%)
fmm	6.0	862.1	1345.6	364.0	963.1	3534.9 (34%)
freqmine	5.5	852.7	1307.3	363.3	948.0	3471.4 (33%)
streamcluster	6.1	875.3	1362.9	365.7	961.4	3565.5 (36%)
vips	5.7	878.8	1339.6	370.6	968.4	3557.4 (34%)
Baseline						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	137.5	1766.9	1628.1	776.7	2001.2	6172.9
cholesky	131.2	1492.2	1558.8	624.0	1690.0	5414.4
fmm	123.7	1479.4	1552.6	653.3	1675.7	5360.9
freqmine	119.6	1420.8	1536.6	638.1	1616.6	5212.4
streamcluster	135.4	1546.4	1578.6	682.9	1755.6	5563.6
vips	119.6	1464.4	1566.2	654.9	1667.3	5352.6

In tests for the 64-core system, the Proxy design outperforms baseline design for more than 45% in four out of six benchmarks, and more than 30% in the rest two benchmarks. The time for getting instructions for the baseline design is one order of magnitude larger than that of the Proxy design. Similar to tests for 32-core system, the time for event processing, sending null messages, safety check, and incoming message processing is doubled for the baseline design in *barnes*, *cholesky*, *fmm* and *streamcluster*, and for *vips* and *freqmine* the baseline design has 30% more time in these three categories. And, the Proxy design has around 20% less event processing time than the baseline design in all six benchmarks.

Table 3: Simulation running time in seconds for 64-core model.

Proxy						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	7.0	1487.4	1595.3	620.8	1638.2	5341.6 (45%)
cholesky	10.6	1536.4	2053.3	624.0	1696.7	5910.4 (51%)
fmm	12.4	1537.9	2283.8	618.5	1707.6	6147.7 (47%)
freqmine	2.4	1528.9	1038.7	639.8	1669.9	4877.3 (30%)
streamcluster	9.9	1508.4	1987.2	616.3	1678.0	5789.9 (48%)
vips	1.5	1525.6	930.5	629.2	1641.4	4726.7 (29%)
Baseline						
Benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barnes	159.1	3084.0	1979.8	1292.7	3441.5	9798.0
cholesky	236.6	3752.9	2625.7	1519.8	4159.6	12058.0
fmm	225.6	3567.2	2521.0	1467.8	3974.3	11530.2
streamcluster	203.8	3520.5	2358.3	1439.9	3897.6	11216.3
vips	46.2	2209.6	1110.8	926.7	2452.7	6699.7

In Table 4 we compare the simulation time in seconds for proxy design and baseline design against sequential simulation that only uses one thread. The numbers in brackets are the speedup against the sequential simulation. As we can see, the overall speedup for both design increases with system scale, while the Proxy design consistently outperforms the baseline design and shows better scalability than the baseline design. In tests for all system models, Proxy has 42% to 104% better speedup, which no doubt is a significant improvement.

Table 4: Comparison with Serial Simulation.

16-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	15711.9	3409.1 (4.6x)	4871.8 (3.2x)
cholesky	17906.1	3415.1 (5.2x)	6042.8 (3.0x)
fmm	16594.0	3389.3 (4.8x)	6588.7 (2.5x)
freqmine	17811.0	3306.2 (5.4x)	5905.7 (3.0x)
streamcluster	17690.1	3379.5 (5.2x)	5870.7 (3.0x)
vips	18327.1	3376.4 (5.4x)	6380.7 (2.9x)
32-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	26038.1	3623.2 (7.2x)	6172.9 (4.2x)
cholesky	25321.0	3649.9 (6.9x)	5414.4 (4.7x)
fmm	24777.1	3534.9 (7.0x)	5361.0 (4.6x)
freqmine	24252.3	3471.4 (7.0x)	5212.4 (4.6x)
streamcluster	23643.6	3565.5 (6.6x)	5563.6 (4.3x)
vips	24139.4	3557.4 (6.8x)	5352.6 (4.5x)
64-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	60487.7	5341.6 (11.3x)	9798.0 (6.2x)
cholesky	64611.1	5910.4 (10.9x)	12058.0 (5.4x)
fmm	73299.5	6147.7 (11.9x)	11530.2 (6.4x)
freqmine	38808.0	4877.3 (8.0x)	7013.6 (5.5x)
streamcluster	63508.5	5789.9 (11.0x)	11216.3 (5.7x)
vips	34195.6	4726.7 (7.2x)	6699.7 (5.1x)

4.2.2 *Analysis of Results*

As mentioned above the only difference between the two designs is how the core-cache LPs get instructions, and intuitively performance gap, if any, should be mainly determined by the difference of time it takes to get instructions, which is also the communication latency between the front-end and the back-end. However, as shown in Tables 1, 2, and 3, the difference in communication latency does not account for the overall performance gap, and that difference alone only contributes to a fraction of the difference in total simulation time between the two designs. The effect of communication latency can spread to other parts of the timing simulation, with higher communication latency the time for safety check, sending null messages, and processing incoming messages grows correspondingly in the baseline design. The reason we found is that, when communication latency increases, core-cache LPs that run out of instructions stall for longer time without making progress and could have an impact on other core-cache LPs that have dependency, causing them to make less progress.

Figure 4 shows the count for core-cache LPs entering different sections of the simulation during a 50 million cycle simulation. As we can see, the counts for doing event processing that actually moves the overall simulation forward is identical for both designs. However, simulations with the baseline design enter the sections for sending null messages, safety check, and processing incoming messages much more often than the Proxy design, and as system scale increases these three counts grow sharply for the baseline design. Tremendous increase in counts means more time spent in these activities for the baseline design, during which the simulation does not move forward.

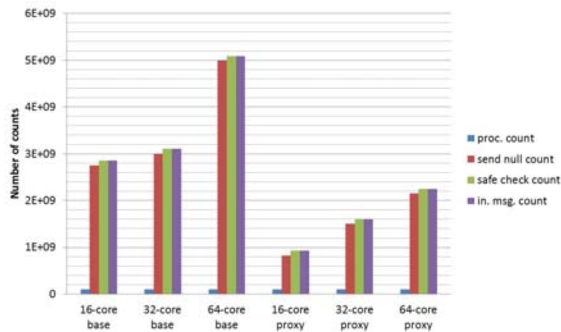

Figure 4: Counts for simulation entering different execution sections.

5. CONCLUSIONS

In timing-directed simulation of multi-core systems with full-system model, how efficiently the timing back-end acquires instructions for its architecture components from the functional front-end is critical for overall simulation performance. In comparison with our baseline design that connects the back-end models directly to the QSim Server in the front-end, our new Proxy design proposed in this paper achieved much better performance. The reason for such performance gap can be traced to the Proxy design's success in hiding the TCP/IP latency between the front-end and the back-end. The reduced overhead in the communication with the front-end seems to have a much larger effect than the reduced overhead itself. Our study has shown that this reduced overhead also affects other activities of null message-based timing-directed simulation of multi-core systems. These effects together contribute to a significant improvement of performance. The proxies, along with QSim server, form an efficient front-end with negligible overhead. For our future work, we plan to create a distributed version of QSim server which would remove TCP/IP communication with the back-end, but would still adopt ideas presented in this paper to reduce inter-process communication overhead.

Acknowledgment

This work is supported by the National Science Foundation, under grant CNS-855110.

6. REFERENCES

[1] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: Infrastructure for full system simulation. *ACM SIGOPS Operating Systems*, 43(1):52–61, January 2009.

[2] R. Bedicheck. Simnow: Fast platform simulation purely in software,. *In Hot Chips 16*, Aug 2004.

[3] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[4] Z. Dong, J. Wang, S. Yalamanchili, and G. Riley. A study of the effect of partitioning on parallel simulation of multicore systems. *IEEE 21st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'13)*, Aug 2013.

[5] http://www.qnx.com. clock_gettime. `http://www.qnx.com/developers/docs/6.4.1/neutrino/lib_ref/c/clock_gettime.html`.

[6] Intel. Pin - a dynamic binary intrumentation tool. `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[7] Intel. Qemu, a fast and portable dynamic translator. *In USENIX 2005 Annual Technical Conf.*, pages 41–46, Apr 2005.

[8] C. Kersey, A. Rodrigues, and S. Yalamanchili. A universal parallel front-end for execution driven microarchitecture simulation,. *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools*, p(p):25–32, 2012.

[9] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. *International Symposium on Performance Analysis of Software and Systems*, pages 53–64, 2009.

[10] J. Miller, H. Kasture, G.Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[11] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B.Jacob. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, March 2011.

[12] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel Distrib. Technol.*, 3(4):34–43, 1995.

[13] M. S.Papamarcos and J. H.Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ISCA '98 25 years of the international symposia on Computer architecture*, pages 284–290, 1998.

[14] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili. Manifold: A parallel simulation framework for multicore systems. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[15] J. Wang, Z. Dong, S. Yalamanchili, and G. Riley. Optimizing parallel simulation of multicore systems using domain-specific knowledge. *2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM SIGSIM PADS)*, 2013.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *in 22nd Annual International Symposium on Computer Architecture.*, pages 24–33, 1995.