SIP server performance on multicore systems

C. P. Wright
E. M. Nahum
D. Wood
J. M. Tracey
E. C. Hu

This paper evaluates the performance of a popular open-source Session Initiation Protocol (SIP) server on three different multicore architectures. We examine the baseline performance and introduce three analysis-driven optimizations that involve increasing the number of slots in hash tables, an in-memory database for user authentication information, and incremental garbage collection for user location information. Wider hash tables reduce the search time and improve multicore scalability by reducing lock contention. The in-memory database reduces interprocess communication and locking. Incremental garbage collection smooths out peaks of both central processing unit and shared memory utilization, eliminating bursts of failed SIP interactions and reducing lock contention on the shared memory segment. Each optimization affects single-core performance and multicore scalability in different ways. The overall result is an improvement in absolute performance on eight cores by a factor of 16 and a doubling of multicore scalability. Results somewhat vary across architectures but follow similar trends, indicating the generality of these optimizations.

Introduction

Multicore processors have emerged as the norm for computers that range from laptops to high-end servers. A multicore processor combines multiple independent processing cores on a single chip. Each core has its own functional units, which allows the cores to operate in parallel. Cores on the same chip share access to memory and input/output devices. Current multicore processors typically have two to four cores per chip, and this number is expected to grow [1].

For decades, improvements in processor performance resulted largely from increasing the clock frequency. However, a small increase in frequency typically yields an even smaller increase in performance but causes a large jump in power consumption. In one typical example, an increase in clock speed of 20% improved performance by only 13% and increased power consumption by 73% [2]. The substantially increased power consumption, resulting from years of increasing the clock frequency, has caused power concerns to become the limiting factor in chip design [3]. This has motivated designers to pursue other approaches to improve performance, such as exploiting thread-level parallelism.

Multicore processors are a direct result. Instead of continuing to increase the clock frequency, designers decreased the frequency by about 20%. This reduced power consumption by roughly a factor of two, which allowed the processing core to be duplicated on the same chip [4]. The net effect is a substantial (though less than twofold) increase in processing power with only modest increases in power consumption and design complexity.

Of course, the multicore approach has its own difficulties. Harnessing multiple cores requires workloads that can be processed in parallel. If a workload lacks sufficient parallelism, cores simply remain idle. Some workloads consist of a large number of independent tasks that can easily be run in parallel. Others that consist of a small number of tasks have successfully been decomposed into multiple parallel subtasks. Some workloads are inherently serial and are difficult to effectively code for multicore systems.

Network workloads tend to be well suited to multicore processors due to the inherent parallelism in processing individual network packets, particularly across multiple connections. This paper presents an analysis of one specific network workload, i.e., the Session Initiation Protocol (SIP), on three different multicore architectures, one from each of AMD, Intel, and IBM.

SIP is an Internet standard protocol for establishing and managing multimedia sessions. It is a protocol of growing importance with uses in Voice over Internet Protocol (VoIP), instant messaging, IP television, and

Digital Object Identifier: 10.1147/JRD.2009.2036976

voice and video conferencing. Many VoIP providers such as Vonage and Gizmo use SIP, as do digital VoIP offerings from traditional telephone companies such as Verizon and AT&T, as well as cable companies such as Time Warner and Comcast. It is also the basis for the IP Multimedia System standard for the Third Generation Partnership Project. Industry analyst reports show that North American VoIP adoption is growing by millions of subscribers a year [5].

The performance characteristics of SIP are not yet as well understood as traditional workloads such as transaction processing and Web serving. In addition, SIP has quality-of-service (QoS) requirements that make its performance characteristics more complex than other protocols.

To evaluate the performance of an SIP server, our approach is to subject the server to a realistic workload that simulates a large number of telephony users. We analyze the performance of the server using a number of techniques, including code profiling and resource monitoring. We identified several modifications to the SIP server software that collectively yield substantial improvements in performance and, more importantly, scalability in terms of how well the cores are utilized. First, we increased the size of a synchronized hash table used to store user location information from 2^9 to 2^{17} entries. Second, we eliminated the use of a conventional database (DB) by instead using a simple in-memory DB for user location and password information. Third, we implemented incremental garbage collection (GC) to smooth peaks in both memory and CPU utilization. Collectively, these optimizations improve performance by 8.7 times for one core and 16.6 times for eight cores.

This paper provides two main contributions. The first contribution is a comprehensive performance analysis of a realistic SIP workload on three different multicore architectures. The second is a set of techniques that substantially improve multicore performance and scalability for the SIP workload.

The remainder of this paper proceeds as follows. First, we provide additional background on SIP. Then, we describe our experimental setup and present our results in detail. Next, we present related work. Finally, we give our conclusions.

Background

This section provides a short background on SIP. More details can be found in [6].

Protocol overview

SIP is a protocol designed to create, modify, and destroy media sessions between two or more endpoints. Request for Comments (RFC) 3261 [7] is the core SIP specification, although many additional RFCs extend the protocol. (An RFC is a memorandum published by the Internet Engineering Task Force describing information related to the Internet and Internet-connected systems.) Sessions can be of various types, including voice, video, and text. SIP handles control messages (e.g., those involved with creating the session) and is referred to as a control-plane protocol, but the transfer of the actual session data is handled by a separate data-plane protocol. SIP runs over the Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Stream Control Transmission Protocol, Secure Sockets Layer, and Internet Protocol versions 4 and 6. SIP is not a network resource reservation protocol such as RSVP [8]; the issue of how to allocate and manage network bandwidth and transfer media is outside the scope of the protocol. SIP is a text-based protocol that borrows many of its mechanisms from the Hypertext Transfer Protocol (HTTP) [9]. Messages contain headers and, optionally, bodies, depending on the message type.

As an example, for voice applications, SIP messages encapsulate an additional protocol, i.e., the Session Description Protocol [10], which provides an offer/answer model to negotiate session parameters, such as which voice codec to use. Once the endpoints agree to the session characteristics, voice data are typically carried by the Real-Time Transport Protocol (RTP) [11]. Media protocols usually directly communicate between endpoints in a peer-to-peer fashion.

SIP user agents and servers

SIP users are usually identified using an SIP Uniform Resource Identifier (URI), e.g., *sip:user@domain.com*. This provides a layer of indirection that enables features such as location independence and mobility. SIP endpoints are typically referred to as user agents, which initiate and accept sessions. They include both hardware (e.g., VoIP phones, cell phones, and pagers) and software (e.g., software phones, instant message clients, and voicemail servers). User agents are further categorized into user agent clients (UACs) and user agent servers (UASs). A UAC initiates a transaction by making a request (e.g., making a call or registering a location), whereas a UAS responds to a request. Within the same call, a user agent may take on both roles (e.g., if the callee hangs up or disconnects, it is the UAC during the call termination). Thus, most call flows for SIP messages define how the UAC and UAS behave for that scenario. Here, the term *callee* refers to the function, subroutine, or person being called by the caller.

The SIP infrastructure includes two crucial server components. The first component involves *registrars* that provide the service for users to register their location with the SIP infrastructure. Users can provide multiple locations. Proxies consult with a registrar to determine a user's address.

The second component involves *proxies* that route messages toward their eventual destinations. If a message is intended for a user for which the proxy is responsible for (i.e., is in its domain), the registrar is queried to determine the final destination for the message. If the proxy is not responsible for the user, the message is forwarded to another proxy. The “next hop” proxy can be chosen via multiple methods, but a common approach is via domain name system lookup [12].

The above services are *functionally* described. In practice, they can be colocated. For example, in our experiments, the server is both a proxy and a registrar.

Transactions
SIP consists of several layers that define how the protocol is functionally composed but not necessarily implemented. One such layer is the *transaction* layer, which is responsible for matching each response to the corresponding request and managing SIP application-layer protocol timeouts and retransmissions. All SIP endpoints (i.e., user agents) have transaction layers, as do stateful proxies. Stateless proxies do not have such layers.

SIP uses HTTP-like request/response transactions. A transaction consists of a request to perform a particular method (e.g., `INVITE`, `BYE`, or `CANCEL`, etc.) and at least one response to that request. Responses may be *provisional*, namely, they provide some short-term feedback to the user (e.g., `100 TRYING` and `180 RINGING`) to indicate progress, or they can be *final* (e.g., `200 OK` and `401 UNAUTHORIZED`). Final responses complete the transaction, whereas provisional responses do not.

A *transaction stateful proxy* is a proxy that maintains state for each transaction that passes through it, for both the client and the server. It performs hop-by-hop retransmission and generates provisional responses (e.g., `100 TRYING`) to inform the previous hop that the proxy has received the message and takes responsibility for retransmitting it. A *stateless proxy*, on the other hand, simply acts as an SIP message forwarder, does not guarantee delivery, and does not generate provisional responses.

The transaction state can last seconds and even minutes, depending on the scenario. The transaction state must be maintained across messages in global shared memory that is accessible by any process or thread, since there is no guarantee that the original servicing process or thread will handle subsequent messages for that transaction. How the memory is shared depends on the server architecture (e.g., process based or event driven), but fundamental to this sharing is the use of locking primitives to synchronize access and ensure consistent data.

Authentication and security
SIP security is complex, as described in the RFC [7], and is an area of ongoing research [13]. Describing all the approaches to SIP security is outside the scope of this paper, but we describe SIP authentication since it is used in our experiments. SIP servers may respond to requests with a challenge for the calling user agent to authenticate itself. This is done via a `401 UNAUTHORIZED` response (for registrars) or a `407 UNAUTHORIZED` response (for proxies), which presents a challenge to which the UAC must respond. The response includes an `Authorization:` header that provides a credential derived from an Message-Digest algorithm 5 (MD5) hash of a nonce (unique identifier), realm, username, and password [14].

The key user information is thus the user ID and password, which are each used for authentication. This user information must also be available for concurrent access in a shared state that is protected by locks.

SIP registration
A user agent notifies the SIP infrastructure where it can be located via registration. This allows a proxy to determine where messages for a particular URI should be routed. Section 2.1 of RFC 3665 [15] shows a packet call flow example for SIP registration with authentication enabled. A client sends a message to the registrar containing its SIP URI, contact IP address, and port number and an expiration time that specifies for how long the information is valid. The registrar then challenges the client to prove its identity using the `401 UNAUTHORIZED` response. The client then re-retransmits the `REGISTER` message with an additional `Authorization:` header that provides the appropriate credentials. The registrar then checks the response and, if correct, saves the contact information for that client’s SIP URI and responds with a `200 OK` message. If authentication is not enabled by the registrar, the packet flow simply consists of the `REGISTER` message from the client followed by the `200 OK` message from the registrar. This case might occur in an enterprise environment where the network is secured and users are trusted.

The key information maintained by the SIP server in this case is the user ID and the corresponding port and IP address. This must again be available for parallel access in a shared state guarded by locks.

SIP proxying
Proxying forwards an SIP message toward its eventual destination in the SIP infrastructure. **Figure 1** shows the SIP call flow for stateful proxying (without authentication). The first SIP client sends an `INVITE` message to the proxy to establish a session with the second SIP client. Since the proxy is stateful, it responds with a `100 TRYING` message to inform the client that the message has been received and that the client need not worry about hop-by-hop retransmissions. The proxy then consults the registrar for the SIP URI of the second client and, assuming it is available, forwards the message. The second client, in turn, acknowledges the receipt of the message and informs the proxy that it is notifying the user via the `180 RINGING` message. The proxy then forwards that message to the initiator of the `INVITE`, informing the client that the end host has received the message and that the

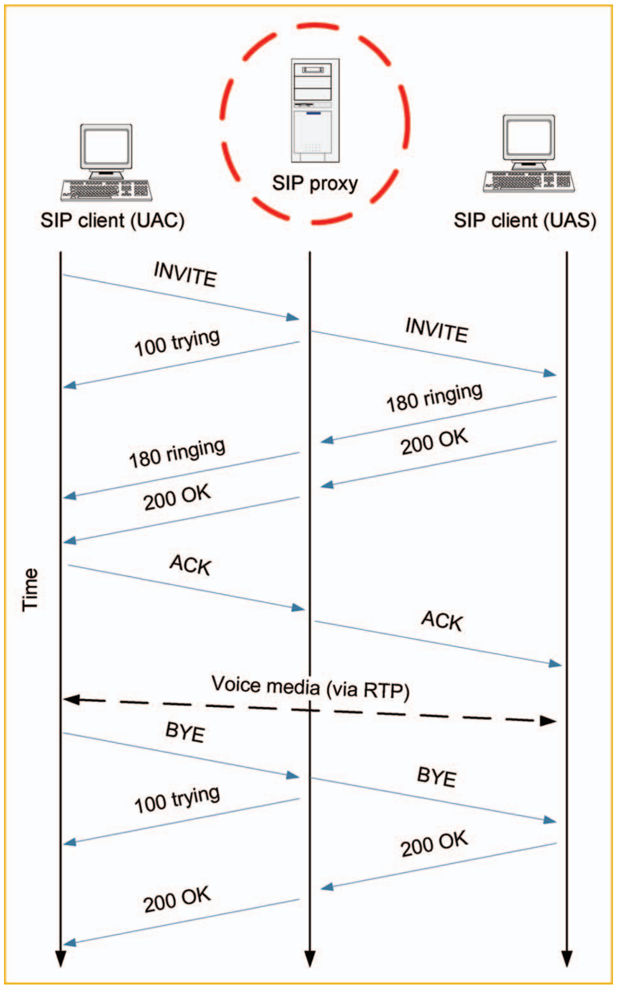

Figure 1

Stateful proxying call flow. (UAC: user agent client; UAS: user agent server.)

line is "ringing." The user on the second client machine then accepts the call, generating a message, which is sent to the proxy, which forwards it on to the first client. The first client then generates an acknowledgment. Having established the session, the two endpoints directly communicate, i.e., in a peer-to-peer fashion, using a media protocol such as RTP [11]. By design, however, the media session does not traverse the proxy. In this paper, we focus solely on the control-plane behavior of an SIP server, and thus, the media are not included in our tests. When the conversation is finished, the first user "hangs up" (disconnects) and generates a response that the proxy forwards to the second user. The second user then responds with a `200 OK`, which is forwarded back to the first user.

The above scenario describes stateful proxying without authentication. If authentication were enabled, the proxy would challenge the UAC `INVITE` and `BYE` requests with `407 UNAUTHORIZED` responses, requiring the UAC to retransmit the requests with the proper credentials (not shown in the figure).

Proxying results in numerous transactions being executed on the proxy server. As described earlier, these transactions require a shared state that must be guarded by locks. One key function that any proxy must perform is user lookup. When a request is received for an SIP URI, the proxy must resolve that URI into a port and IP address in order to forward the request to the proper host. This lookup also requires concurrent access to a shared state that must be protected by locks.

Experimental setup

In this section, we describe the software and hardware utilized in our experiments.

Workload and metrics

The main benchmark we use is a preliminary version of the Standard Performance Evaluation Corporation (SPEC**) SIP infrastructure benchmark [16]. SPEC is a nonprofit entity that establishes and publishes various standardized benchmarks. We briefly describe SPEC SIP here.

In essence, the SPEC SIP benchmark is a user-based benchmark that models a number of virtual users who register their location with the proxy–registrar and make phone calls to one another. The benchmark is an SIP control-plane benchmark and includes no media. Increasing the load is accomplished by instantiating more virtual users. Users behave statistically rather than deterministically. For example, they have variable "think" times before making calls, variable ring times before answering a call, and variable hold times for talking on the phone. (The "think time" is the time that a user is idle between making calls.) This benchmark is much more sophisticated than is usually the case in other SIP benchmarking studies, which typically have no ring time, no hold times (or fixed hold times), and no user lookup.

We parameterized the benchmark software to match the current SPEC SIP workload definition when we ran the benchmarks in late 2008. For values that were not yet defined (e.g., runtime), we chose sensible values based on our experience with the benchmark and an array of SIP servers. Registration is relatively simple: each user registers every 10 minutes with a 15-minute timeout at the registrar. All registrations are authenticated. Phone calls are more complex. Each user makes a phone call using an exponentially distributed think time of 1 hour (e.g., the "busy hour call attempt," i.e., a call attempted while the server is at its busiest), and the user talks (the "hold time") for a variable amount of time depending on the call type. There are three call types. Regular calls, called *completed calls* by the benchmark, have hold times that are log-normally distributed with an average of 300 seconds

Table 1 Benchmark server processors.

Type	*Processor*	*Clock speed*	*Cores*	*L2 cache*
AMD	Opteron 8218	2.6 GHz	Four dual core	1 MB/one core
Intel	Xeon X5450	3.0 GHz	Two quad core	6 MB/two cores
IBM	POWER6	4.0 GHz	Two dual core, two contexts/core	4 MB/one core

and make up 65% of the call volume. Voicemail calls, or calls routed to voicemail when the callee is unavailable, are 30% of the call volume with an exponentially distributed average call hold time of 30 seconds. Canceled calls, which make up the remaining 5% of the call volume, are calls in which the caller hangs up before the call is accepted by the callee. Thus, each instantiated user makes a call similar to the flow depicted in Figure 1. All `INVITE` and `BYE` messages are authenticated.

SPEC SIP has certain QoS requirements that must be met for a run to be considered valid. In our benchmarks, 99.99% of SIP transactions must be successful. When a server is overloaded, transactions can fail because either the overloaded server sends an unexpected response (e.g., timeout) or the maximum number of retransmissions is exceeded. Thus, the metric for the SPEC SIP benchmark is the simultaneous number of conforming users, which represents the number of virtual users instantiated by the benchmark. The peak throughput is the largest number of users that the system can support while maintaining the QoS metrics. We use the term users as a shorthand. UDP is used as the transport, and a test run lasts for 2 hours after a 10-minute warm-up time. A warm-up time is simply used to eliminate any transients that may occur when the system is first booted.

Hardware end connectivity

We use three server processor architectures in our experiments, in order to examine the impact of processor architecture on our results. Each system has a total of eight processing cores and thus appears to the operating system as an eight-way system.

The first is an IBM LS41 blade server with four 2.6-GHz dual-core AMD Opteron** 8218 Santa Rosa [17] processors. Each core has a dedicated 1-MB L2 cache. It also has two Broadcom NetXtreme** II BCM5706S Copper Gigabit interfaces.

The second system is an IBM xSeries* 3500 server with two 3.0-GHz quad-core Intel Xeon** X5450 Harpertown processors. Each pair of cores shares a 6,144-KB L2 cache. It also has two Broadcom NetXtreme II BCM5721 Copper Gigabit interfaces.

The third is an IBM JS22 blade server with two 4-GHz dual-core POWER6* [18] processors, each with two symmetric multithreading (SMT) contexts. Each core has a 4-MB L2 cache. It also has two IBM eHEA Gigabit Ethernet devices, where HEA stands for host Ethernet adapter.

Table 2 Client workload generation machines.

Type	*Quantity*	*Processor*	*Clock speed*	*RAM*
LS21	20	AMD Opteron	2.6 GHz	4 GB
x346	15	Intel Xeon	3.4 GHz	5 GB
HS20	10	Intel Xeon	3.0 GHz	4 GB
x3650	2	Intel Clovertown	2.0 GHz	10 GB

All of the machines had 16 GB of RAM and Small Computer System Interface disks. Each machine had two network interfaces. One was connected to the local area network of our site, while the other is connected to a private network dedicated to the benchmark. To minimize experimental perturbation and variability, all of our measurements were conducted using the private network, where minimal other traffic occurs. The server machines are summarized in **Table 1**.

Due to the large amounts of load we must generate, we use up to 47 machines of four types as client workload generators. For brevity, the systems are summarized in **Table 2**. Each client machine has two Gigabit Ethernet interfaces appropriately connected. We generate such large loads to ensure that the server is the performance bottleneck and that no client or network capacity limitations influence our measurements.

Server software

We use the Open SIP Express Router (OpenSER) [19], a commonly used freely available open-source SIP proxy–registrar server. OpenSER has a large feature set and a considerable user base, and it is associated with an active mailing list and third-party contributions (e.g., from sip.edu and onsip.org). We use OpenSER version 1.2.3 for our experiments. We produced a configuration file that supported stateful proxying and registration. In situations in which a user DB was required, we used MySQL** [20] 5.0.45-7.el5, which we precisely populated with the required number of users for a given test. OpenSER was configured to use a writeback caching policy to maintain the client state across restarts and also to nearly achieve in-memory DB performance. For the POWER6 port of OpenSER, we discovered and fixed a locking bug that caused crashes at high loads. OpenSER is the predecessor of the OpenSIPS [21] and Kamailio [22] SIP servers.

Client workload generation
We use the SIPp [23] SIP workload generator, another freely available open-source tool. SIPp is also associated with an active user community and appears to be the most frequently used open-source SIP testing tool. SIPp allows a wide range of SIP scenarios to be tested, such as scenarios involving UACs, UASs, and third-party call control. SIPp is also extensible by writing Extensible Markup Language (XML) scripts that define new call flows. We wrote several new flows that were not included with SIPp to implement the SPEC SIP benchmark. SIPp has many runtime options we took advantage of, such as MD5-based hash digest authentication and support to allow calls to be generated from a list of users.

We use SIPp SVN (subversion) release version r541. During the course of this study, we made several modifications to SIPp to improve its performance so as to reduce the amount of client resources. All of these extensions have been incorporated into the current SIPp SVN tree.

Client and server operating system software
Our servers run Red Hat** Enterprise Linux** AS Release 5 update 2. On the AMD and Intel servers, this includes a 2.6.18-92.1.1.el5 kernel. On the POWER6 server, a 2.6.18-92.1.10.el5 kernel is used. The client machines use either Red Hat EL AS Release 5 update 2 or Red Hat EL AS Release 4 update 4.

For application and kernel profiling, we use the standard open-source OProfile [24] tool, version 0.9.3. OProfile is configured to report the default `GLOBAL_POWER_EVENT`, which reports the time in which the processor is not stopped (i.e., idle time is not reported).

Results
In this section, we present several performance and scalability bottlenecks in OpenSER that we identified and corrected on an eight-way AMD LS41 blade. We also present performance measurements of a baseline OpenSER server and a server with our optimizations on eight-way Intel and POWER6 machines.

OpenSER baseline
A baseline OpenSER configuration on one core supports 350,000 users. Adding a core improves performance by 28%, resulting in 450,000 users, and eight cores yield only an additional 150,000 users or an overall improvement of 71% when eight times the processing resources are available.

We define multicore scalability for a configuration as the ratio of the performance on a given number of cores to the performance of the same configuration on one core. Thus, the two-core case has a scalability of 1.28, or 128%, and the eight-core case has a multicore scalability of 1.71, or 171%.

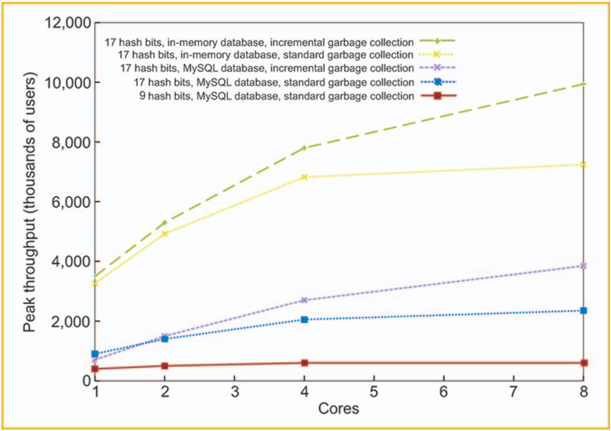

Figure 2

OpenSER optimizations on an AMD LS41 blade.

As the default OpenSER configuration only uses four processes, it is not surprising that scalability is limited for higher numbers of cores. Thus, we increased the number of OpenSER processes to eight. An additional statically configured limit to performance is the size of the shared memory region of OpenSER, which is used to ensure that the independent worker processes have coherent transaction and user location states. We increased the size of this memory region to 12 GB, which still leaves 4 GB remaining for the per-process private memory of OpenSER, MySQL, the operating system, and other processes that run on the machine. These results are shown in **Figure 2** with the label "9 hash bits, MySQL database, standard garbage collection." The maximum capacity increased by 50,000 users for one and two cores. For four cores, the maximum capacity increased by 33% to 600,000 users, and the capacity was identical to that for eight cores. We use these options (i.e., eight OpenSER processes and 12 GB of shared memory) in all further tests, because they are essential for the scalability of other configurations, and we wish to remove their effect from the analysis of other optimizations.

Hash table lookup
We used the OProfile whole-system profiler to gain insight into the behavior of the system. A summary of the profiles at peak throughput is shown in **Figure 3**. We categorized the profiled functions by modules (MySQL, Kernel, OpenSER, C Library, and Other). We further subdivided OpenSER into the time spent in the `get_urecord` function and all other times. The `get_urecord` function accounts for between 11.9% and 16.2% of the total profile or 42.6%–54.7% of the CPU cycles spent in OpenSER. The `get_urecord` function

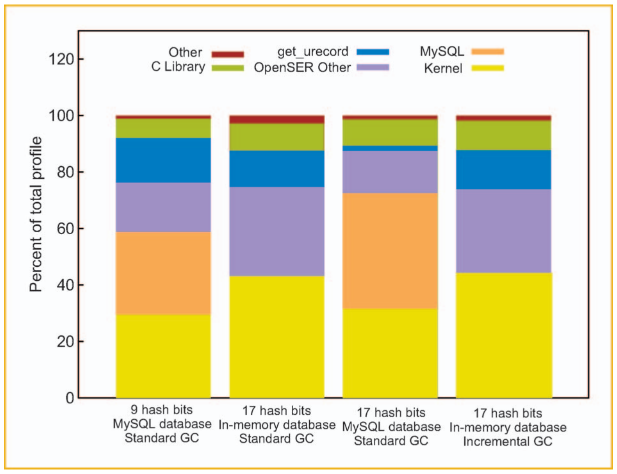

Figure 3

OpenSER CPU profiling. Standard GC uses the GC mechanism of OpenSER. Incremental GC uses our improved GC mechanism described in the section on incremental GC.

is used to retrieve a user's location information (sent via registration) from the in-memory user location cache of OpenSER. The cache is stored as a 512-bucket hash table using chaining to resolve collisions. As we are storing between 400,000 and 600,000 users in these configurations, the average chain length is roughly 1,000 users. Not only is this linear scan CPU intensive, but it also limits concurrence since each hash chain is protected by a lock. We increased the hash table size (via an OpenSER configuration parameter) to 2^{17} entries, labeled as "17 hash bits, MySQL database, standard garbage collection" in Figure 2. Our new profile shows that only 1.1%–1.9% of the time is now spent in the `get_urecord` function. Reducing this time significantly improved performance by 2.6, 3.1, 4.5, and 3.9 times for one, two, four, and eight cores, respectively. In addition to improving absolute performance, multicore scalability improved as well: from 1.25 to 1.55 for two cores, from 1.50 to 2.27 for four cores, and from 1.50 to 2.61 for eight cores.

A better solution than simply configuring OpenSER with a fixed number of hash bits would be to modify OpenSER to automatically tune its hash table size based on the number of entries. This could be implemented using one of several well-known methods for dynamically sizing hash tables, including linear hashing or spiral storage [25].

Database access

The next avenue of research that we explored was converting DB accesses to in-memory data structures. In the original profile, MySQL used 30%–37% of the CPU time, and in the new profile, the use was in excess of 40% of the CPU cycles. Moreover, we know from prior work that a significant portion of the C Library computational cycles can be attributed to the use of MySQL [26]. In our configuration, OpenSER uses the MySQL DB for two distinct functions. First, it is used as a repository for user location information (i.e., the port and IP address where a user can be reached). Second, it is used to store user authentication information (i.e., usernames and passwords). Eliminating the use of the DB for user location is straightforward; OpenSER provides an entirely in-memory user location mode (essentially, the DB cache is loaded upon start-up and never flushed).

Eliminating the authentication DB is not as simple. OpenSER is structured such that the authentication module depends on a DB module. Although a flat-text "database" module (called dbtext) is included with OpenSER, it is designed for demonstration purposes and is not suitable for use in a scalable SIP server. The OpenSER DB interface that dbtext follows requires the interface to be a general-purpose DB access application programming interface, with arbitrary tables and column names, which are parsed out from the text file. Moreover, each dbtext access locks the entire in-memory representation of the DB while linearly scanning it. Our module, which we call `auth_mem`, is derived from the higher level OpenSER `auth_db` module. It is relatively simple, consisting of only 240 lines of new code that loads a Comma Separated Values file of username–password pairs on start-up, and replaces calls to back-end DB modules with an in-memory hash table lookup (we used a width of 2^{17}, as we did in the user location module).

The results of removing the DB can be seen in Figure 2 as the curve labeled "17 hash bits, in-memory database, standard garbage collection." To isolate the effects of each DB, we also evaluated removing them individually and in combination; however, for clarity, we do not display the results in the graph. Compared to the 17-hash-bit MySQL DB configuration, in-memory user location improved performance by 33% for one core, 57% for two cores, 53% for four cores, and 34% for eight cores. In-memory authentication increased performance by 76%, 90%, 34%, and 19% for one, two, four, and eight cores, respectively. We would expect that eliminating the authentication MySQL DB would provide a greater improvement than eliminating the MySQL user location DB. The user location DB includes an in-memory cache that is updated by registration and consulted by other transactions (e.g., `INVITE` or `BYE`). The MySQL DB is only updated every minute during a periodic writeback phase and is only read upon start-up. However, the authentication DB has no caching, so each SIP transaction must consult the DB. As expected, removing the authentication DB improved performance more for one and two cores; however, for four and eight cores, removing the user location DB provided a greater improvement.

The combined performance improvement is a factor of 3.6, 3.5, 3.3, and 3.1 for one, two, four, and eight cores.

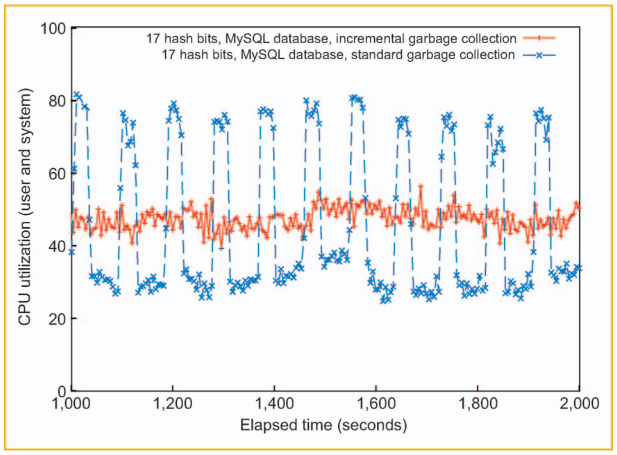

Figure 4

OpenSER CPU utilization timeline for four cores and 800,000 users.

Interestingly, removing both DBs improves performance more than the sum of individually removing each DB. As the number of cores increases, the relative performance increase is reduced, resulting in worse scalability. This demonstrates that simply improving performance does not necessarily increase multicore scalability. The largest reduction in scalability is for eight cores, which went from 2.6 with MySQL to 2.2 with the in-memory DB. In many deployments, it may not be practical to entirely eliminate traditional DBs; for example, in clustered servers, the DB server can act as a coherency mechanism. Nevertheless, these results serve as an upper bound of the gains possible from optimizing DB access. One possibility is to cache the authentication information from the DB in-memory; however, care must be taken to maintain coherency between the cache and the DB (otherwise, revoked credentials could still be valid).

Incremental garbage collection

The final optimization we identified concerns improving the garbage collection behavior of OpenSER. An in-memory hash table of user location entries is stored in a shared memory segment, for both the DB and entirely in-memory configurations. As SIP is a soft-state protocol, these entries periodically expire and must be pruned.

During our test, we used *nmon* [27] to collect a variety of system performance information every 5 seconds. Using these data, we generated timelines, and when focusing on a small region of the timeline, we noticed that the CPU utilization had a periodic behavior. An example of one of these timelines is shown as the curve "17 hash bits, MySQL database, standard garbage collection" in **Figure 4**. We noticed that the gap between each CPU spike was roughly 60 seconds, and we correlated this to a timer with that same period that is responsible for pruning expired records from the OpenSER shared memory segment. When this timer is triggered, OpenSER iterates through all of the records in shared memory and frees those that are no longer needed. This behavior has two deleterious effects on performance. First, the CPU utilization is quite high during this scan period, and thus, calls are more likely to fail. Second, because shared memory can often be a bottleneck and this method leaves expired entries in shared memory longer than necessary, unnecessary call failures can result from shared memory allocation requests that fail.

We improved this behavior by changing the OpenSER GC so that it ran every second but only cleaned 1/60 of the shared memory region. This does not reduce the total amount of work, nor does it significantly increase the amount of work. In fact, in this example, the average CPU utilization is within 1% of the original value. However, this change to GC does more uniformly spread the work. As can be seen in Figure 4, the CPU utilization is much smoother in the improved curves labeled "17 hash bits, MySQL database, incremental garbage collection." We also analyzed the same timelines for shared memory, and we noticed that shared memory usage was also markedly smoother.

This approach translated into actual performance improvements, as shown in Figure 2. For a single core with a DB, peak performance decreased by 22% but increased by 7.1%, 31.7%, and 63.8% for two, four, and eight cores, respectively. With the in-memory DB, performance was improved for all numbers of cores: 7.3%, 7.6%, 14.3%, and 37.2% for one, two, four, and eight cores, respectively.

Collectively, on the AMD architecture, our optimizations improved performance by 8.7 times for a single core and 16.6 times for eight cores. This results in scalability improving from 1.5 to 2.84 on eight cores.

Other architectures

Our next set of experiments compare the effect of our optimizations on SIP server performance across three architectural platforms described in the section on hardware and connectivity: the Intel, AMD, and POWER6 processors. In this section, we focus on scalability as the metric.

Figure 5 shows the peak throughput for each of our three platforms for two configurations each: the baseline (before applying our optimizations) and optimized (after applying all optimizations discussed in the prior sections on hash table lookup, DB access, and incremental GC) configurations. As can be seen in the figure, the optimizations improved absolute performance in all cases ranging from a factor of 4.5 to a factor of 20.8. We now turn our attention to scalability.

Table 3 presents the eight-core scalability for each platform and configuration. We are interested in observing how scalable the system is before and after applying our optimizations. Scalability is defined here as the relative

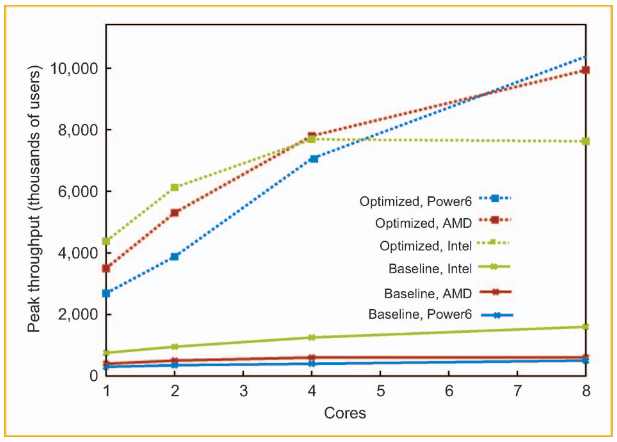

Figure 5

Optimizations on multiple architectures: AMD, Intel, and IBM POWER6.

Table 3 Eight-core scalability.

Processor	*Baseline*	*Optimized*
AMD Opteron	1.50	2.83
Intel Xeon	2.13	1.74
IBM POWER6	1.66	3.86

performance compared to the single-core case; thus, by definition, each value for a single core is 1. The most striking result is the difficulty in achieving linear speedup: the highest scaling is only 3.86 with eight processors, for the optimized POWER6 curve. The lowest corresponds to the baseline AMD curve, with a speedup of 1.50.

Another interesting result is that for the baseline configurations, scalability was relatively low in all cases. Intel is the only platform with a scalability greater than 2 (2.13). As no platform scaled particularly well, there was no change in the order of relative performance as the number of cores changed. For the optimized case, scalability varied much more widely, from 1.74 on the Intel platform to 3.86 on the POWER* platform. Indeed, the relative performance on eight cores is opposite the performance on a single core. The POWER platform was fastest on eight cores and slowest on a single core, and the Intel platform was fastest on a single core and slowest on eight.

Our optimizations improved absolute performance for all architectures, but our optimizations were clearly more effective at improving scalability for the AMD and POWER platforms than for the Intel platform. Although none of these optimizations are processor specific, we gathered profiles for these optimizations on an AMD platform; thus, it is not surprising that the AMD platform would derive greater benefit than another platform. One similarity between AMD and POWER chips is that they both provide dedicated L2 caches for each core, whereas Intel does not. Additionally, the AMD and POWER chips integrate the memory controller in the CPU, whereas the Intel memory controller is off chip and accessed via the front-side bus. Thus, while the Intel platform is, in some respects, superior to AMD and POWER platforms in the optimized single-core case by 25% and 63%, respectively, it is 30% and 36% slower in the eight-core case. It is also interesting to note that Intel was almost three times as fast as POWER and AMD in the eight-core baseline case but was slower for the optimized case. To better understand this, we applied only the 17-hash-bit optimization and found that the relative performance gap was significantly reduced, with Intel having an improvement of only 10% over AMD and 33% over POWER for eight cores.

Related work

Due to space limitations, we only briefly mention related work in the SIP server performance area.

Multicore SIP server performance

Perhaps the most closely related SIP work is that of Zou et al. [28]. They also study multicore scalability of the OpenSER SIP server, not only on a Linux on Intel platform, but also using Solaris** on Niagara, a system developed by Sun Microsystems. They additionally consider TCP as a transport protocol. They encountered scalability bottlenecks in both the Linux and Solaris operating systems, such as those caused by a single lock protecting access to a socket, and propose using multiple sockets (and port numbers) in response to avoid the problem. They also noticed a problem involving contention in the shared memory segment used by OpenSER. Their response is to hash a shared state by call ID and use multiple shared memory segments to partition the global state and have a lock per segment, reducing the contention. They improved scalability by up to a factor of four, depending on the scenario.

Our work, in contrast, did not encounter these problems. This is primarily because our workload is user based, rather than a scale-up of a simple microbenchmark. Thus, the bottlenecks we encounter tend to be more focused on scaling structures to support users. We also observed problems resulting from insufficient shared memory space, but our solution was simply to increase the segment to 12 GB.

Multicore Web serve performance

Ruan et al. [29] examine Web server performance using a Linux system and using several servers on an Intel Xeon hyperthreaded processor. They find that network server workloads do not scale well on the Xeon processor due to insufficient cache resources available to each thread.

Veal and Foong [30] examined Web server scalability using Linux, Apache, SPECWeb2005**, and a server with two Intel Clovertown quad-core processors. After considering and dismissing several potential bottlenecks, they conclude that the address bus is the primary obstacle to performance scaling of network applications.

Willmann et al. [31] compare two approaches to network protocol stack parallelization using a TCP microbenchmark. They find that parallelizing at the connection level performs better than parallelizing at the message level.

Zeldovich et al. [32] present an asynchronous programming library to enable parallelized event-driven programs. They show a 50% speedup on a nonstandard Web benchmark using four cores.

Single-core SIP server performance

Nahum et al. [26, 33] study SIP server performance in several scenarios: registration and proxying, transaction stateful versus stateless, and using UDP or TCP for the transport. They show that performance significantly varies depending on how the SIP server is configured and used.

Ram et al. [34] extend the above work, showing how to improve the performance of OpenSER when using TCP. Janak's thesis [35] describes many of the performance optimizations that are used by the SIP Express Router (and, by implication, OpenSER). Examples include using counted strings, lazy parsing of headers, and incremental parsing within each header. Salsano et al. [36] conduct an experimental performance analysis of SIP security mechanisms using an open-source Java** SIP proxy server. Cortes et al. [37] measured the capacity of four transaction stateful SIP proxies using a suite of five microbenchmark tests.

SIP benchmarking

The SPEC SIP benchmark [16] is described in the section on workload and metrics. SIPStone [38] is an early SIP benchmark designed to evaluate SIP registrar, redirect, and proxy servers. The benchmark is the weighted average of ten SIP microbenchmark call flows.

Multicore network processors

A recent trend has been to utilize multiple processors for core network workloads such as routing and deep packet inspection. Examples are discussed in [39–46].

Conclusion

We have evaluated and improved the performance and scalability of an open-source SIP proxy on three different multicore platforms: AMD Santa Rosa, Intel Harpertown, and IBM POWER6. We have identified two performance and scalability bottlenecks using whole-system profiling:

- OpenSER relied on a statically dimensioned hash table for user location lookup. By increasing the width of this hash table, we have improved performance by a factor of four. Additionally, scalability improved by 121%.
- DB accesses require context switches and socket operations, which reduce performance. By changing DB accesses to use memory, performance increased by a factor of 3. However, scalability was slightly reduced from 2.6 to 2.2 with eight cores.

We have identified a third optimization by examining CPU utilization timelines and correlating the peaks with timers in the OpenSER code. Here, OpenSER performed GC for the entire shared memory region at once. This resulted in alternating periods of very high CPU and shared memory utilization and periods of much lower utilization. These resource utilization spikes would then result in call failures. By incrementally garbage collecting the shared memory region, we have eliminated these spikes and increased performance up to 63%.

Collectively, our optimizations improved performance by 8.7 times for a single core and 16.6 times for eight cores. This results in an eight-core scalability improvement from 1.5 to 2.84.

When comparing CPU architectures, note that the single-core performance did not predict the relative eight-core performance for our optimized version of OpenSER. In fact, the performance ranking for a single core was the opposite of the ranking for eight cores.

One result manifested throughout this work is that improving the single-core performance is not the same as improving scalability. Often, improving the performance actually makes it more difficult to scale the single-core case, since system resources that were not previously a bottleneck become constrained. We have also noticed that there are many static configuration parameters for SIP servers. For example, to achieve acceptable performance, OpenSER must have the user location hash table and shared memory region correctly sized by the system administrator. OpenSER is not alone in this regard; for example, Java-based servers must have correctly sized heaps. For the system to be scalable in practice, it must adapt to the load to which it is subjected. In the case of OpenSER, this means that its data structures must scale in near-constant time and that the shared memory region should grow and shrink along with the load. One mechanism to achieve this effect would be to use threads and heap space instead of processes and shared memory.

Future work

Several avenues of research exist that would help to further improve the scalability of OpenSER. Our analysis mainly relied on whole-system CPU profiling of CPU cycles and CPU utilization. Performing a similar analysis using lower level counters to gather the cycles per instruction, as well as cache hits and misses, would help expose scalability problems. Moreover, this would allow us to analyze the varying effect of our optimizations on different architectures. We also did not focus on lock contention. OpenSER locks are very similar to spin locks in the Linux kernel. This means that employing a tool or methodology similar to `lockmeter` may highlight lock contention. In addition, our analysis focused entirely on OpenSER. OpenSER is dependent on the performance of both the DB and the kernel; improvements in either could yield significant capacity gains. Finally, we studied a proxy–registrar workload that models an enterprise environment on a particular SIP server. Changing either the workload or the system under tests will yield new scalability limitations that must be addressed.

Acknowledgment

The authors would like to thank L. Shao for his useful comments.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of the Standard Performance Evaluation Corporation, Advanced Micro Devices, Inc., Broadcom Corporation, Intel Corporation, MySQL AB, Red Hat Software, Inc., Linus Torvalds, or Sun Microsystems, Inc., in the United States, other countries, or both.

References

1. P. Gepner and M. Kowalik, "Multi-core processors: A new way to achieve high system performance," in *Proc. Int. Symp. Parallel Comput. Elect. Eng.*, Sep. 2006, pp. 9–13.
2. J. Rattner. (2006). "Why multi-core?" in *Intel Developer Forum*. [Online]. Available: http://www.intel.com/pressroom/kits/events/idfspr_2006/20060307_rattnertranscript.pdf
3. R. Ronen, A. Mendelson, K. Lai, S. Lu, F. Pollack, and J. Shen, "Coming challenges in microarchitectures and architecture," *Proc. IEEE*, vol. 89, no. 3, pp. 325–339, Mar. 2001.
4. S. Naffziger, "High-performance processors in a power-limited world," in *Proc. IEEE Symp. VLSI Circuits*, Honolulu, HI, Jun. 2006, pp. 93–97.
5. P. Arora. (2001, May). *Residential Gazing Into the Crystal Ball of Tomorrow's Residential Voice Services, Analyst Report, Frost & Sullivan*. [Online]. Available: http://www.frost.com/prod/servlet/market-insight-top.pag?Src=RSS&docid=97662447
6. R. Sparks, "SIP: Basics and beyond," *Queue*, vol. 5, no. 2, pp. 22–33, Mar. 2007.
7. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. J. ston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session initiation protocol,* Internet Engineering Task Force, RFC 3261, Jun. 2002.
8. L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Netw. Mag.*, vol. 7, no. 5, pp. 8–18, Sep. 1993.
9. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext Transfer Protocol—HTTP/1.1,* Internet Engineering Task Force, RFC 2068, Jan. 1997.
10. J. Rosenberg and H. Schulzrinne, *An offer/answer model with session description protocol (SDP),* Internet Engineering Task Force, RFC 3264, Jun. 2002.
11. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RTP: A transport protocol for real-time applications,* Internet Engineering Task Force, RFC 3550, Jul. 2003.
12. J. Rosenberg and H. Schulzrinne, *Session Initiation Protocol (SIP): Locating SIP Servers,* Internet Engineering Task Force, RFC 3263, Jun. 2002.
13. Internet Engineering Task Force (IETF), *SIP Working Group Charter*. [Online]. Available: http://www.ietf.org/html.charters/sip-charter.html
14. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, *HTTP Authentication: Basic and Digest Access Authentication,* Internet Engineering Task Force, RFC 2617, Jun. 1999.
15. A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers, *Session Initiation Protocol (SIP) Basic Call Flow Examples,* Internet Engineering Task Force, RFC 3665, Dec. 2003.
16. Systems Performance Evaluation Corporation (SPEC), *SPEC SIP Subcommittee*. [Online]. Available: http://www.spec.org/specsip/
17. C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, Mar./Apr. 2003.
18. H. Q. Le, W. J. Starke, J. S. Fields, F. P. OConnell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM J. Res. & Dev.*, vol. 51, no. 6, pp. 639–662, Nov. 2007.
19. *The Open SIP Express Router (OpenSER)*. [Online]. Available: http://www.openser.org
20. The MySQL Project, *The MySQL Database Server*. [Online]. Available: http://www.mysql.org
21. *OpenSIPS (Open SIP Server)*. [Online]. Available: http://www.opensips.org
22. *Kamailio—The Open Source SIP Server*. [Online]. Available: http://www.kamailio.org
23. R. Gayraud and O. Jacques, *SIPp*. [Online]. Available: http://sipp.sourceforge.net
24. OProfile, *A System Profiler for Linux*. [Online]. Available: http://oprofile.sourceforge.net/
25. P. Larson, "Dynamic hash tables," *Commun. ACM*, vol. 31, no. 4, pp. 446–457, Apr. 1988.
26. E. Nahum, J. Tracey, and C. P. Wright, "Evaluating SIP proxy server performance," in *Proc. 17th Int. Workshop NOSSDAV*, Urbana-Champaign, IL, Jun. 2007.
27. N. Griffiths, *nmon Performance: A Free Tool to Analyze AIX and Linux Performance*. [Online]. Available: http://www.ibm.com/developerworks/aix/library/au-analyze_aix/
28. J. Zou, Z. Liang, and Y. Dai, "Scalability evaluation and optimization of multi-core SIP proxy server," in *Proc. 37th ICPP*, Portland, OR, Sep. 2008, pp. 43–50.
29. Y. Ruan, V. S. Pai, E. M. Nahum, and J. M. Tracey, "Evaluating the impact of simultaneous multithreading on network servers using real hardware," in *Proc. ACM Sigmetrics Conf. Meas. Model. Comput. Syst.*, Banff, AB, Canada, Jun. 2005, pp. 315–326.
30. B. Veal and A. Foong, "Performance scalability of a multi-core web server," in *Proc. ACM/IEEE Symp. ANCS*, Orlando, FL, Dec. 2007, pp. 57–66.
31. P. Willmann, S. Rixner, and A. L. Cox, "An evaluation of network stack parallelization strategies in modern operating systems," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, Jun. 2006, p. 8.
32. N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and M. F. Kaashoek, "Multiprocessor support for event-driven programs," in *Proc. USENIX Annu. Tech. Conf.*, San Antonio, TX, Jun. 2003, pp. 239–252.
33. E. Nahum, J. Tracey, and C. P. Wright, "Evaluating SIP server performance," IBM T.J. Watson Res. Center, Yorktown Heights, NY, Research Rep. 24183, Feb. 2007.
34. K. K. Ram, I. C. Fedeli, A. L. Cox, and S. Rixner, "Explaining the impact of network transport protocols on SIP proxy performance," in *Proc. IEEE ISPASS*, Austin, TX, Apr. 2008, pp. 75–84.
35. J. Janak, "SIP server proxy effectiveness," M.S. thesis, Dept. Comput. Sci., Czech Tech. Univ., Prague, Czech Republic, May 2003.

36. S. Salsano, L. Veltri, and D. Papalilo, "SIP security issues: The SIP authentication procedure and its processing load," *IEEE Netw.*, vol. 16, no. 6, pp. 38–44, Nov./Dec. 2002.
37. M. Cortes, J. R. Ensor, and J. O. Esteban, "On SIP performance," *Bell Labs Tech. J.*, vol. 9, no. 3, pp. 155–172, 2004.
38. H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, *SIPstone—Benchmarking SIP Server Performance*, Apr. 2002. [Online]. Available: http://www.sipstone.org
39. J. Giacomoni, J. K. Bennett, A. Carzaniga, D. C. Sicker, M. Vachharajani, and A. L. Wolf, "Frame shared memory: Line-rate networking on commodity hardware," in *Proc. ACM/IEEE Symp. ANCS*, New York, Dec. 2007, pp. 27–36.
40. X. Hu, X. Tang, and B. Hua, "High-performance IPv6 forwarding algorithm for multi-core and multithreaded network processor," in *Proc. 11th ACM SIGPLAN Symp. PPoPP*, New York, Mar. 2006, pp. 168–177.
41. D. Liu, B. Hua, X. Hu, and X. Tang, "High-performance packet classification algorithm for many-core and multithreaded network processor," in *Proc. Int. Conf. CASES*, Seoul, Korea, Oct. 2006, pp. 334–344.
42. A. Mallik, Y. Zhang, and G. Memik, "Automated task distribution in multicore network processors using statistical analysis," in *Proc. ACM/IEEE Symp. ANCS*, Dec. 2007, pp. 67–76.
43. P. Piyachon and Y. Luo, "Efficient memory utilization on network processors for deep packet inspection," in *Proc. ACM/IEEE Symp. ANCS*, New York, Dec. 2006, pp. 71–80.
44. R. Smith, D. Gibson, and S. Kong, "To CMP or not to CMP: Analyzing packet classification on modern and traditional parallel architectures," in *Proc. ACM/IEEE Symp. ANCS*, Dec. 2007, pp. 43–44.
45. Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards high-performance flow-level packet processing on multi-core network processors," in *Proc. ACM/IEEE Symp. ANCS*, Dec. 2007, pp. 17–26.
46. Y. Qi, B. Xu, F. He, X. Zhou, J. Yu, and J. Li, "Towards optimized packet classification algorithms for multi-core network processors," in *Proc. 36th ICPP*, Xi'An, China, Sep. 2007, p. 2.

Received December 8, 2008; accepted for publication January 11, 2009

Charles P. Wright *IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (cpwright@us.ibm.com).* Dr. Wright received the B.S., M.S., and Ph.D. degrees from the State University of New York at Stony Brook in 2003, 2004, and 2006, respectively. He subsequently joined the IBM Thomas J. Watson Research Center, where he worked in the Network Server Systems Software Department investigating SIP workload generation and server performance and is currently a Research Staff Member. Recently, his focus has changed to high-performance computing. He is the author or a coauthor of 21 technical papers.

Erich M. Nahum *IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (nahum@us.ibm.com).* Dr. Nahum received the B.A. degree in computer science from the University of Wisconsin–Madison in 1988 and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst, in 1991 and 1996, respectively. He subsequently joined IBM at the T. J. Watson Research Center, where he has worked on networked systems performance and is a currently a Research Staff Member in the Computer Sciences Department. In 2000, he received an IBM Outstanding Technical Achievement Award for his work on AIX* server performance. He is the author or a coauthor of 32 technical papers. He is the holder or a coholder of three patents.

David Wood *IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (dawood@us.ibm.com).* Mr. Wood received the B.A. degree in physics from the University of California, San Diego, La Jolla, in 1985 and the Master's degree in computer science from New York University, New York, in 1989. He works at the IBM T. J. Watson Research Center, where he focuses on policy technologies and parallel computing. He has been with IBM since 1992, working in areas that include parallel computing, data visualization, visual and spoken user interfaces, and location-based services and architectures. He is the author or a coauthor of seven papers and journal articles. He is the co-holder of five patents.

John M. Tracey *IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (traceyj@us.ibm.com).* Dr. Tracey received the B.S. degree in electrical engineering and the M.S. and Ph.D. degrees in computer science from the University of Notre Dame, Notre Dame, IN, in 1990, 1992, and 1996, respectively. In 1996, he joined IBM as a Software Engineer at the IBM T. J. Watson Research Center, Hawthorne, NY, where he is currently a Senior Technical Staff Member and Manager. From 2001 to 2008, he managed the Network Server System Software Department, which focused on performance analysis and improvement of various network servers. In 2009, he began managing a new team developing system software for massively parallel systems. He is a coauthor on more than a dozen technical publications. He is a coholder of seven issued patents.

Elbert C. Hu *IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (elbert@us.ibm.com).* Mr. Hu received the B.A. degree in computer science from Queens College, City University of New York, New York, in 1979 and the M.S. degree in computer science from the Polytechnic Institute of New York University, Brooklyn, in 1981. He joined the IBM Research Division in 1982 and has been involved in projects that focused on VM/Control Program, TCP/IP, network management, distributed computing, high-performance computing, and SIP performance. He was the recipient of three Research Division Awards, an IBM Outstanding Technical Achievement Award for his work on Advanced Fast Path Architecture in IBM HTTP server products, and an IBM Outstanding Contribution Award for his work on IBM VM and MVS* TCP/IP products.