CommBench - A Telecommunications Benchmark for Network Processors

Tilman Wolf
Mark Franklin

WUCS-99-29

November 11, 1999

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130

CommBench - A Telecommunications Benchmark for Network Processors

Abstract

This paper presents a benchmark, *CommBench,* for use in evaluating and designing telecommunications network processors. The benchmark applications focus on small, computationally intense program kernels typical of the network processor environment. The benchmark is composed of eight programs, four of them oriented towards packet header processing and four oriented towards data stream processing. The benchmark is defined and various characteristics of the benchmark are presented. These include instruction frequencies, computational complexity, and cache performance. These measured characteristics are compared to the SPEC benchmark which has traditionally been used in evaluating workstation processors. Three examples are presented indicating how *CommBench* can aid in the design of a single chip network multiprocessor.

1 Introduction

In recent years the telecommunications industry has been expanding rapidly. Conservative estimates for aggregate bandwidth on the internet backbone indicate a doubling each year for the past ten years, and further expansion at these levels is likely to continue for some time to come [6]. One consequence of this grow is the demand for greater performance, flexibility, reliability and cost effectiveness in the primary electronic components (e.g., routers and switches) which control the flow of data through the network.

At the system level, these networks have used computer control techniques for some time. Performance demands, however, now include not only bandwidth requirements, but also quality of service, encoding/decoding of packets, dynamic bandwidth management and routing, and intelligent error recovery. The trend is to develop highly flexible routers which, in some sense, are programmable "on the fly." Programmability can range from utilizing predefined functions which can be applied to packets traversing the router, to 'active' networks where packets contain, or dynamically invoke, programs that are executed on the router [25].

These new requirements have pushed processing activities down to the network layer of the system. Traditionally, bit level operations, like timing synchronization and error detection, as well as packet level operations, like routing and switching, have been implemented in dedicated logic to keep up with the speed of the incoming traffic. However, as processor performance has increased, the domain of “real-time” activities which can be placed under program control has grown.

The advantages of using processor cores instead of hard-wired logic include being able to specialize processing at the lowest levels in response to customer needs, being able to modify processing in response to discovery of errors and in response to changes in telecommunications standards, and being able to have a single resuable core processor within a chip instead of multiple custom logic designs. This flexibility can be significant when dealing with a product line involving many chips since it impacts design, documentation and testing costs.

Until recently, most embedded processors for telecommunications have been standard microprocessors (e.g., MIPS, Power PC, ARM). Generally, these processors have been designed and optimized for workstation and personal computer environments and have, for example, extensive floating point features. A standard benchmark used in their evaluation has been the SPEC suite [24]. However, the tasks associated workstations and PCs differ from those associated with telecommunications and it is important to have a benchmark that reflects this different environment.

This paper presents a set of benchmark applications, called ***CommBench,*** tailored to the telecommunications environment. The rationale for selecting these programs is discussed, the benchmark, and an extensive set of measurements are presented and contrasted with those obtained using the SPEC integer benchmark. Differences and network processor design implications are discussed. Note that the issues associated with benchmark selection are often controversial and, in the longer term, industry associations and input are important. However, the field is evolving so rapidly that we feel it is important to move ahead with a benchmark proposal. It is anticipated that refinements will be necessary and will be part of an ongoing process. Nevertheless having a benchmark is important now when research and industry are still in the relatively early stages of developing network processor designs.

The following section presents background material and related work. In Section 3 the benchmark applications and data sets are described. Section 4 contains the bulk of the paper and considers a set of measurements on both *CommBench* and SPEC. Section 5 gives three examples on how the benchmark results can influence network processor design. Section 6 summarizes the contributions of this work.

2 Background and Related Work

In response to the increasing cost and performance demands associated with telecommunications applications, a number of alternative strategies to developing embedded processors are being considered. While some companies have chosen to develop their own proprietary processors, the general trend has been to use commercially available core processors. These processors constitute a portion of an ASIC chip where the remainder of the chip area contains dedicated logic and memory associated with a given telecommunications product.

While the basic design of the core processor typically follows standard advanced RISC design principles, companies producing and marketing core processors [11] [16] often provide for some specialization. Specialization features fall into the following general categories:

- Memory Organization: Selection of cache memory size, associativity and design.
- Word Size: Selection of register/instruction word size (e.g., 16, 24 or 32 bit).
- Functional Components: Selection of a limited set of predesigned functions (e.g., floating point, timers).
- Coprocessor Functions: Standard coprocessor interfaces for the to perform special functions [12]. The coprocessor, generally designed by the company purchasing the processor core, acts as a special logic block which is invoked by the processor [28].
- New Instructions: Provision for the establishment of new instructions which execute within the standard RISC pipeline structure [26].

This set of alternatives represents the first generation of choices associated with the development of reconfigurable processor designs. While the above approaches are static in nature, dynamic reconfiguration is becoming more feasible and dynamically reconfigurable instruction sets and pipelines are likely appear in the next few years [2] [10] [21].

The design issue here is just how to select from the numerous alternatives given the available chip area and performance implications of each decision. There is a long history of developing benchmark programs of both the synthetic (Whetstone [8], Drystone [30]) and real (SPEC [24]) varieties. While the most popular benchmark associated with workstations has been the SPEC integer and floating-point suite, benchmarks aimed towards other application classes have also been developed and used with some success. Two examples are TPC

[27], a benchmark oriented towards transaction processing applications, and SPLASH [32], a benchmark oriented towards scientific applications executing on parallel processors .

In addition to specific differences in program execution characteristics (e.g., cache behavior, etc.), there are other related areas in which traditional benchmarks are inadequate for evaluation of telecommunications oriented processors. One significant shortcoming is the missing focus on clearly defined I/O. For a network environment, a benchmark should ideally consider a wide range of input sizes, permitting analysis of both small (i.e., ATM cells) and large (i.e., data streams) data sizes. Benchmarks such as SPEC also are not focussed on yielding information related to real-time constraints. Additionally, traditional benchmarks tend to assume an applications environment which is relatively static as opposed to the network processor environment where one encounters many small dynamically changing applications.

A recent benchmark, that addresses I/O issues in the context of multimedia applications is the MediaBench benchmark [14] which consists of programs implementing various compression and coding algorithms for streaming voice, audio, and video data (e.g., JPEG, MPEG, GSM, etc.). However, multimedia transcoding is only one part of the network processor applications domain. Additionally, such processors must perform a wide variety of logical control operations not significantly present in MediaBench.

CommBench focuses on operations that are performed at the network layer where applications are heavily dominated by I/O and hard real-time constraints. The benchmark applications are split into two subsets. One considers streaming data flows in a networking context (similar to MediaBench). The other focuses on packet-based processing tasks such as routing, data forwarding and monitoring.

3 The Benchmark

A desirable property of any application in a benchmark is its representativeness of a wider application class in the domain of interest. In this spirit, the benchmark applications have been chosen so that their kernels represent common network processing computations. For example, the tree based lookup in RTR is representative of many routing algorithms as well as packet classification schemes. The discrete cosine transform performed in JPEG is the basis of all JPEG and MPEG coding schemes.

CommBench applications have also been selected to represent typical workloads for both traditional routers (focus on header processing) and 'active' routers (perform both header and stream processing). Thus, the applications can be divided into two groups: ***Header-Processing Applications*** (HPA) and ***Payload-Processing Applications*** (PPA).

3.1 Header-Processing Applications

The header-processing programs represent operations that are done on a per-packet basis and are mainly independent of the size and type of the packet payload. These applications involve a good deal of "random" logic, header field interrogation and processing, table lookup, and control. One issue concerns the selection of programs typical of this domain. At this point, many of the more advanced application programs are embedded into existing network components and are proprietary in nature. We have selected the public domain programs listed below which likely to be operationally similar to proprietary programs.

- ***RTR*** is a Radix-Tree Routing table lookup program. Routing table lookups are important operations performed for every packet in a datagram-based network, and for every connection in a connection-based network. *RTR* is the radix-tree routing algorithm from the public domain NetBSD distribution [17]. There are more efficient routing approaches [23], however they are not freely available.
  Kernel: lookup operations on tree data structure.

- ***FRAG*** is a IP packet fragmentation application. IP packets are split into multiple fragments for which some header fields have to be adjusted and a header checksum computed. The checksum computation that dominates this application is performed as part of all IP packet application programs other than just forwarding.
  Kernel: packet header modifications and checksum computation.

- ***DRR*** is a Deficit Round Robin fair scheduling algorithm [22] that is commonly used for bandwidth scheduling on network links. The algorithm is implemented in one form or another in various switches currently available (e.g., Cisco 12000 series [3]).
  Kernel: queue maintenance and packet scheduling for fair resource utilization.

- ***TCP*** is a TCP traffic monitoring application that is representative of the class of monitoring and management applications. We use *tcpdump*, a widely used tool, that is standard in BSD distributions and is based on the BSD packet filter [15].
  Kernel: pattern-matching on header data fields.

3.2 Payload Processing Applications

Payload-processing applications access and possibly modify the contents of a packet during processing on a network node. Generally the applications are executed not only on a single packet, but on a stream of packets as is common in modern routers. Note that each of these applications has an encoding and a decoding section. While each of these sections is executed separately, they are considered together as a single program unless they have significantly different performance characteristics.

- ***CAST*** is a program based on the CAST-128 block cipher algorithm that uses a 128 bit key to encrypt data for secure transmission [1]. CAST-128 operates similar to other block cipher algorithms used in current networks, such as IDEA [13] and RC5 [19], however, CAST is in the public domain and is not limited by patent restrictions.
  Kernel: encryption arithmetic.

- ***ZIP*** is a data compression program based on the commonly used Lempel-Ziv (LZ77) algorithm [33] for data compression. The implementation can achieve different levels of data compression by varying the algorithm's computational complexity and exemplifies applications that permit tradeoffs between computational power and bandwidth.
  Kernel: data compression.

- ***REED*** is an implementation of the Reed-Solomon Forward Error Correction scheme that adds redundancy to data to allow recovery from transmission errors [18]. This is commonly used on unreliable data links which can be found in wireless networks.
  Kernel: redundancy coding.

- ***JPEG*** is a lossy compression algorithm [29] for image data. It represents the class of media transcoding applications.
  Kernel: discrete cosine transform (DCT) and Huffmann coding.

3.3 Data and Tools

Naturally, the data collected represents executions of the benchmark on a particular processor utilizing a particular compiler. All the programs in the benchmark have been run on SUN UltraSparc II processors operating under the SunOS 5.7. The C compiler used was *gcc* 2.8.1 and was executed with optimization level O2. This level has been selected because the compiler only performs optimizations that are independent of the target processor and does not exploit particular architectural features (e.g., loop unrolling for superscalar machines). To account for the influence that the compiler has on the benchmark characteristics, we compared *gcc* to the *cc* 4.2 compiler. Differences in the generated instruction mix were limited to 1-2% for each instruction class. The cache performance of the generated code was also very similar for both compilers.

For run time instruction mix analysis, Shade [5] and SpixTools [4] were used. These tools simulate and analyze programs on a Sparc processor. For the cache simulations, Dinero [9], a uniprocessor cache simulator, was used.

The benchmark programs were executed with a variety of input data to see the effect on program operation characteristics. While the Header-Processing Applications require data inputs in a particular format for each program (i.e., TCP requires raw packet header, while RTR lookups requires IP addresses), the Payload-Processing Applications, except for JPEG, can process any data stream. For these applications we measured instruction mix and cache behavior for HTML data (plain text), binary program code, and JPEG coded image data.

While CAST and REED perform identically on any data, ZIP shows differences on data that has already been entropy encoded (i.e., JPEG data). To account for this variation, the input for the benchmark measurements was developed with an equal mix of the three data types.

4 Benchmark Characteristics

There is a wide range of characteristics associated with any program or benchmark, and just which of these impacts performance depends on the underlying processor architecture (existing or proposed) and associated compiler. We have selected the following general areas of characterization: code and kernel sizes, computational complexity, instruction frequency, and cache performance. We expect that other characterizations will appear as progress is made towards developing embedded network computers as discussed earlier.

4.1 Code and Computational Kernel Sizes

One can view the size of an application along a number of different dimensions ranging from source code size to the number of bytes most often referenced during execution as shown in Table 1. For *CommBench* and SPEC static and dynamic code size information was collected.

Number of lines of C code	Static
Number of bytes of compiled code	Static
Number of instructions executed at least once	Dynamic
Number of instructions accounting for 90% of execution	Dynamic
Number of instructions accounting for 99% of execution	Dynamic

Table 1: Benchmark Application Sizing

CommBench Program	Type	Code Size C lines	Code Size Object bytes	SPEC Program	Code Size C lines	Code Size Object bytes
TCP	HPA	19,100	352,000	126.gcc	206,000	1,950,000
JPEG	PPA	18,300	260,000	147.vortex	67,200	1,150,000
ZIP	PPA	6,500	117,000	132.ijpeg	31,200	594,000
RTR	HPA	1,130	16,000	099.go	29,200	558,000
REED	PPA	410	6,900	134.perl	26,900	544,000
CAST	PPA	350	19,500	124.m88ksim	19,900	404,000
DRR	HPA	100	2,500	130.li	7,600	139,000
FRAG	HPA	100	2,400	129.compress	1,930	81,700
Average		5,750	97,000	Average	48,700	678,000

Table 2: *CommBench* and SPEC Code Size

The size of the source code and compiled code of each program in both *CommBench* and SPEC is shown in Table 2. The object size data does not include the large but little used dynamically linked libraries (up to 300 kbytes on the SUN Solaris system). *CommBench* programs, based on object code size, are about an order of magnitude smaller than SPEC programs. The variation in *CommBench* code size stems from the different environments in which the applications have been developed. DRR and FRAG are non-commercial proof-of-concept implementations, while ZIP and JPEG are industrial strength implementations with a multitude of options. This has an impact on static code analysis, but dynamic run-time analysis indicates that all applications execute within a fairly small kernel.

The dynamic kernel characteristics of *CommBench* programs are shown in Table 3. The first column indicates the number of instructions which have been executed at least once. Note that the average is 3,430 instructions (13,720 bytes), which is significantly less than the average unlinked object code size (97,000 bytes). Even when one removes from the object code size the roughly 15% which corresponds to data fields, this indicates the presence of a significant amount of code that is never executed. This 'dead' code typically corresponds to code for error handling conditions or rarely used data formats. A similar situation can be seen from the corresponding SPEC data. Defining the ratio, I_c, of instruction code (e.g.,

CommBench Program	Instructions at least once	Instr. for 99%	Instr. for 90%	SPEC Program	Instructions at least once	Instr. for 99%	Instr. for 90%
TCP	7,257	317	232	126.gcc	124,246	43,983	15,899
JPEG	6,155	804	504	147.vortex	60,630	10,136	1,715
RTR	3,805	1,371	387	099.go	53,629	17,511	6,530
ZIP	3,538	555	296	132.ijpeg	12,627	1,735	949
CAST	2,529	716	642	124.m88ksim	12,313	2,154	875
REED	1,510	48	23	134.perl	12,284	683	542
DRR	1,353	70	36	130.li	7,341	990	408
FRAG	1,258	97	80	129.compress	2,842	352	227
Average	3,430	500	275	Average	35,700	9,700	3,390

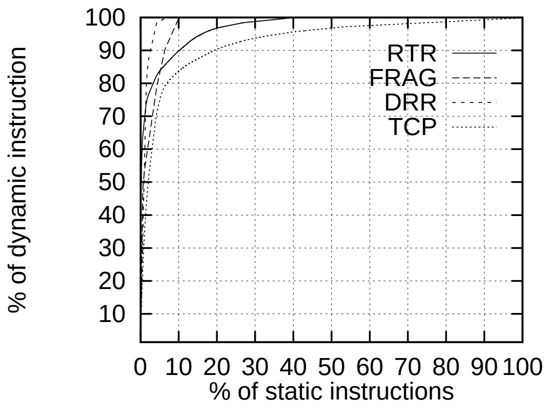
Table 3: *CommBench* and SPEC Dynamic Kernel Characteristics

instructions that have been executed at least once) to the instruction portion of the object code size (e.g., 85% of the object code size), we obtain $I_{c,Comm} = 0.16$ and $I_{c,SPEC} = 0.24$. Thus, not only is the static size for *CommBench* significantly smaller than for SPEC, but the dynamic kernel of *CommBench* is relatively even smaller.

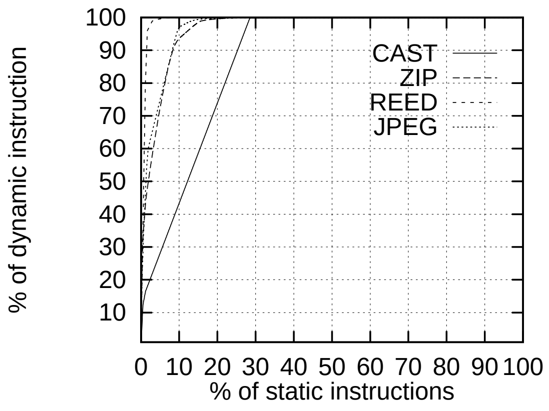
This idea is reinforced by examining the ratio of the number of instructions which constitute 99% of the instructions executed to the number of instructions which are executed at least once, I_k. For *CommBench* $I_{k,Comm} = 0.15$ while for SPEC $I_{k,SPEC} = 0.27$. Additionally, while the ratio of object code sizes for SPEC versus *CommBench* is on average about 7, the ratio of number of instructions which constitute 99% of the executed code is on average almost 20. These results reflect the notion that workstation processors typically execute a few large tasks while network processors can be expected to execute smaller, somewhat simpler but computationally intense tasks. This has significant implications for memory and cache requirements, and will be important when determining the memory requirements for single chip parallel network processors.

A common notion used in processor design is the 90/10 "rule"; that is 90% of executed instructions are derived from 10% of the instructions in the program. Figure 1 is a visual representation of the 90/10 rule showing the size of the kernel in relation to the total number of instructions. A steeply rising curve indicates that only a few instructions are responsible for most of the runtime computation. Only instructions that are executed at least once are considered.

In *CommBench*, RTR, ZIP, and JPEG have kernels that follow the 90/10 rule very closely. FRAG, DRR, and REED have smaller kernels that correspond more to a 95/5 rule. TCP



(a) Header-Processing Applications



(b) Payload-Processing Applications

Figure 1: Kernel characteristics of *CommBench* applications

has a wider spread kernel more closely following a 85/15 rule. CAST has a basically linear behavior due to a fairly large inner loop that repeats many instructions the same number of times.

4.2 Computational Complexity

Given clearly defined I/O components for each *CommBench* application, it is possible to define the computationally complexity of each application with respect to the number and size of the processed packets. This complexity measure helps in determining certain aspects of performance as a function of expected workload. Note, that computational complexity here does not reflect memory system performance since it is based on the number of instructions rather than the number of cycles executed.

We define the computational complexity $N_{a,l}$ to be the number of instructions per byte required for application a operating on a packet of length l. For header processing, l is taken to be 64, 576 and $1,536$ bytes (i.e., minimum IP-packet size, minimum MTU (maximum transfer unit) over IP, and maximum Ethernet packet size). The minimum $l = 64$ is also in the range of ATM cell size (53 bytes). For payload processing applications l is effectively equal to infinity. That is, we consider data streams of sufficient length ($\geq 1 Mbyte$) so that the startup processing overhead is negligible. Table 4 shows the complexity of *CommBench* applications.

Given an average incoming data rate, the results of Table 4 give a preliminary indication of how fast a processor is needed for real-time packet header and payload processing (see example in Section 5.1).

HPA a	$N_{a,64}$	$N_{a,576}$	$N_{a,1536}$	PPA a	$N_{a,\infty}$ (enc)	$N_{a,\infty}$ (dec)
TCP	10.3	1.2	.4	REED	603	1052
FRAG	7.7	.9	.3	ZIP	226	35
DRR	4.1	.5	.2	CAST	104	104
RTR	2.1	.2	.1	JPEG	81	60

Table 4: Computational Complexity of *CommBench* Applications in instructions/byte. Packet sizes for header processing applications are 64, 576 and 1,536 bytes. For payload processing applications the complexity is given as instructions per byte of payload (*enc* =encode/encrypt; *dec* =decode/decrypt)

4.3 Instruction Set Characteristics

The instruction mix gives an indication on the type of instructions executed in the benchmark. Figure 2 shows the instruction set frequencies for each of the programs in *CommBench*. Figure 3 gives the instruction frequencies for *CommBench*, SPEC, and the two *CommBench* components, HPA and PPA. Table 5 presents this same data sorted by frequency for each benchmark. Both, the general trend and the variability across programs is similar to that found in SPEC. The following points out the similarities between the two benchmarks:

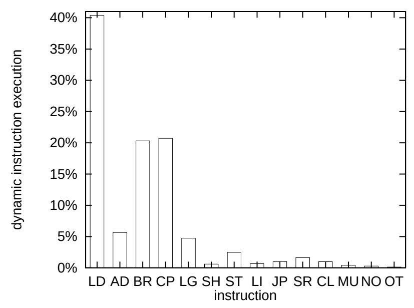
- The *average* difference in frequencies for the top nine instructions ($\approx 97\%$ of executed instructions) between *CommBench* and SPEC is under 5%.
- The eight most frequent instruction types ($\approx 95\%$ of executed instructions) are the same for both *CommBench* and SPEC.

There are also important differences:

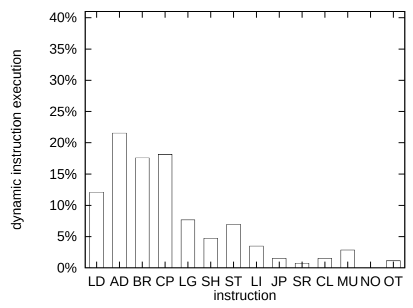
- The *average* variance of the instruction frequencies over all instructions for *CommBench* is 1.4 times that of SPEC. This is due to the fact that the HPA and PPA act as two sub-benchmarks within *CommBench* and have differing execution characteristics.
- *CommBench* executes about 6% more add/sub instructions and 5% fewer load immediate instructions than SPEC.

A comparison of the HPA and PPA benchmark components points to other differences of note.

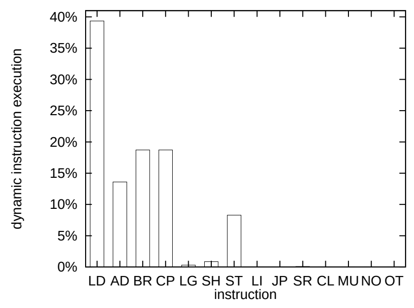
- The two components of *CommBench,* HPA and PPA, have different instruction execution frequencies. For example, there are two instruction groups, the first, including load and



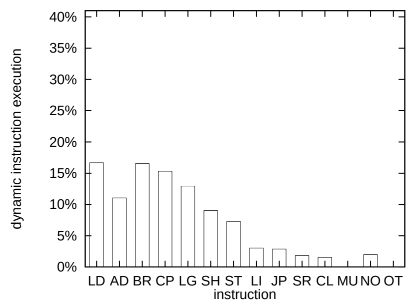
(a) RTR



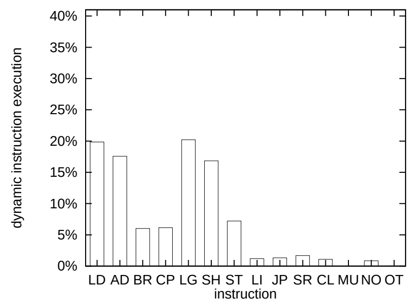
(b) FRAG



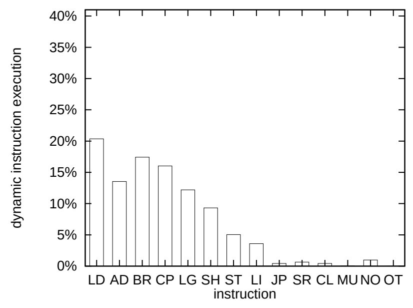
(c) DRR



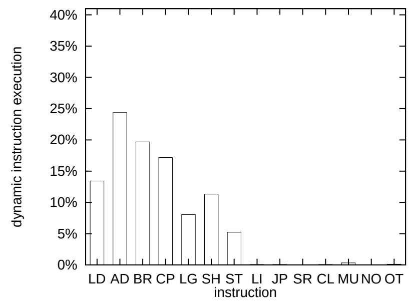
(d) TCP



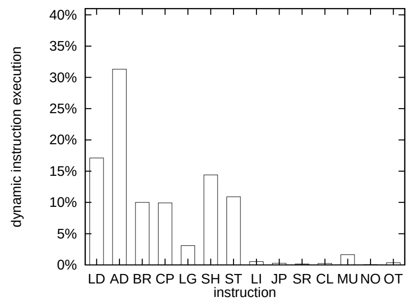
(e) CAST



(f) ZIP

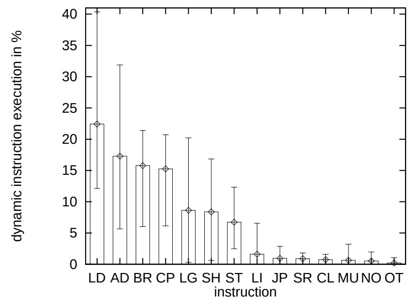


(g) REED

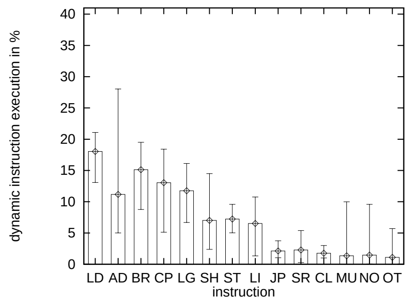


(h) JPEG

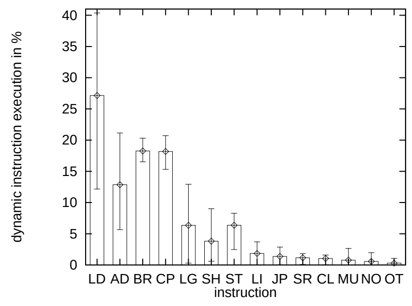
Figure 2: *CommBench* Instruction Frequencies (see Figure 3 for x-axis legend)



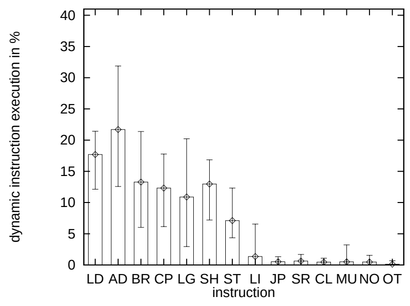
(a) CommBench



(b) SPEC CINT95



(c) CommBench HPA



(d) CommBench PPA

Figure 3: Instruction Frequencies: *CommBench*, SPEC, *CommBench* Header and Payload Processing Application. Error bars indicate the minimum and maximum of instruction frequencies encountered for any single application. (LD = load, AD = add/sub, BR = conditional branch, CP = compare, LG = logic, SH = shift, ST = store, LI = load immediate, JP = jump and link, SR = save/restore, CL = call, MU = mult, NO = nop, OT = other)

CommBench Average	%		SPEC CINT95 Average	%		*CommBench* HPA	%		*CommBench* PPA	%
load	22		load	18		load	27		add/sub	22
add/sub	17		cond. branch	15		cond. branch	18		load	18
cond. branch	16		compare	13		compare	18		cond. branch	13
compare	15		logic	12		add/sub	13		shift	13
logic	9		add/sub	11		store	6		compare	12
shift	8		store	7		logic	6		logic	11
store	7		shift	7		shift	4		store	7
load imm.	2		load imm.	7		load imm.	2		load imm.	1
jmpl	1		save/restore	2		jmpl	1		save/restore	1

Table 5: Instruction Frequencies for *CommBench*, SPEC, HPA and PPA.

add/sub, where the differences in instruction frequencies are about 10%, and the second, including cond branch, compare and logic, where the differences are around 5%.

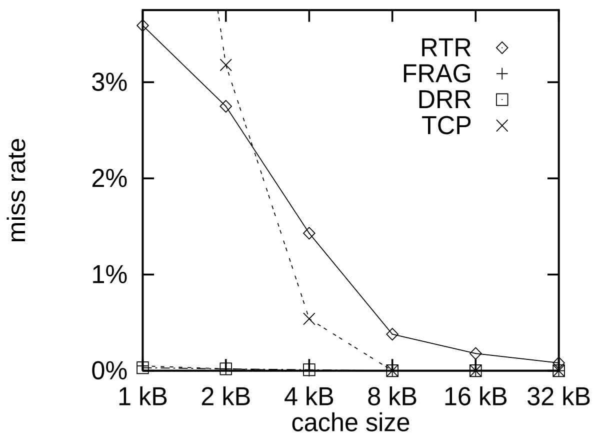
- For certain instructions the differences between SPEC and HPA are significant. For example there are four instructions in the top eight, load, compare, logic and load immediate, where the differences are between 5% and 9%. On the one hand it appears that the SPEC applications are better able to use the load immediate instruction. On the other hand, header processing involves a good deal of load byte from the header into a register and comparing them against some values. This is reflected in *CommBench*'s larger percentages for load and compare.

- For selected instructions, there are significant differences between SPEC and PPA. The frequency difference associated with add/sub is about 11%. In this case the PPA programs have a higher frequency due to the requirements of payload/streaming applications. For other instructions, shift and load immediate, the differences are about 5%.

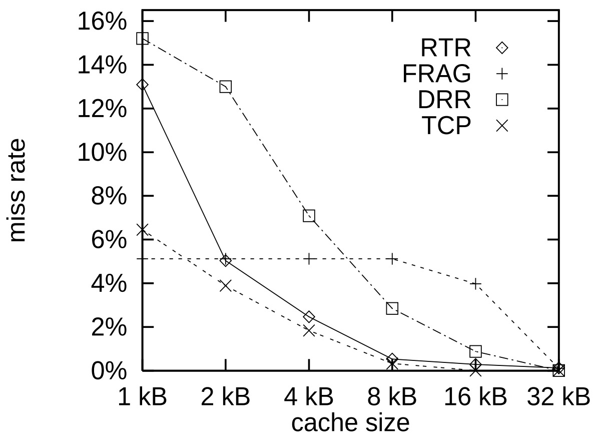
The above differences point out that a network processor must deal with both streaming applications and header processing applications. These results also support a design approach which consists of developing network processors in terms of communicating groups of processing cores where the individual cores have selected characteristics tailored to either HPA and PPA applications. The number of cores of each type within a given chip would generally not be equal since the prior complexity analysis indicates that the processing requirements of the two *CommBench* components are different.

4.4 Memory Hierarchy Characteristics

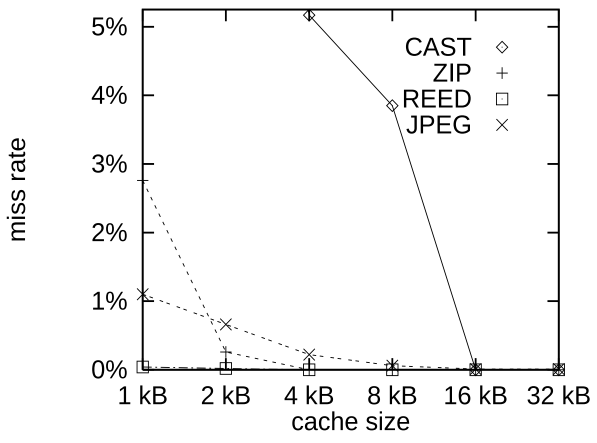
An important part of any processor design is its memory hierarchy. We measured the cache performance for each *CommBench* application. Separate instruction and data caches from 1 kbyte to 32 kbyte were simulated. Figure 4 shows the results for a 2-way associative instruction and data cache. Other caches with different associativity were also investigated. For the direct mapped cache the rule of thumb holds which states that the miss rate is about 1.5 to 2 times that of a 2-way cache. The differences between 2-way, 4-way and 8-way associative caches are minor, hence the gain for going to higher associativity given the additional chip area costs are limited.



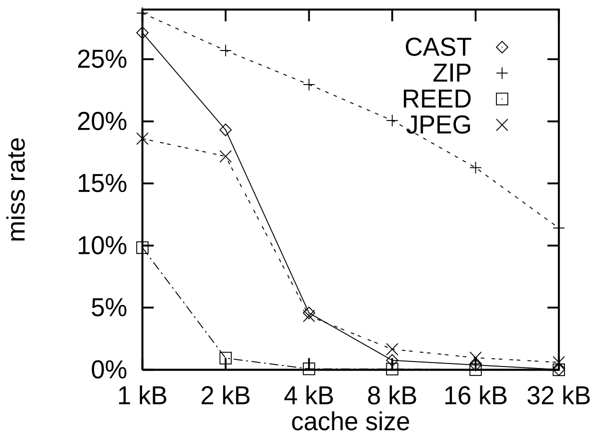
(a) HPA Instruction Cache



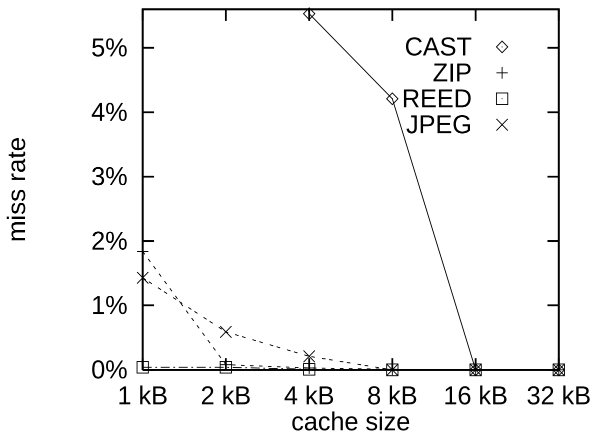
(b) HPA Data Cache



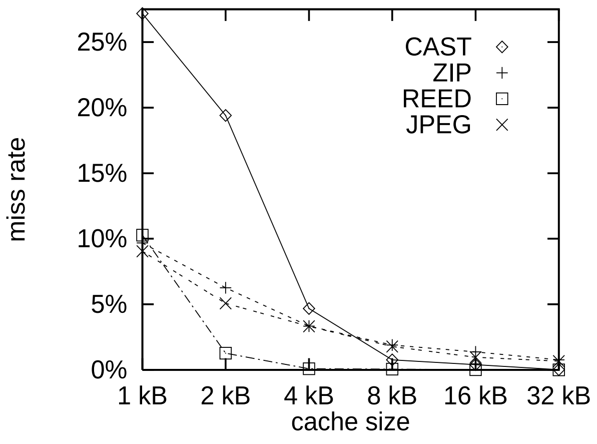
(c) PPA Instruction Cache (Encoding)



(d) PPA Data Cache (Encoding)

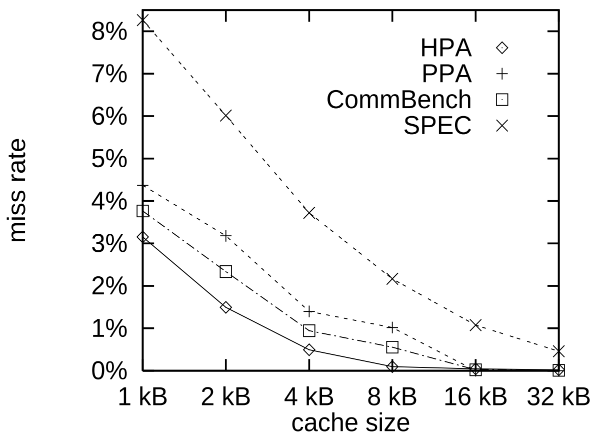


(e) PPA Instruction Cache (Decoding)

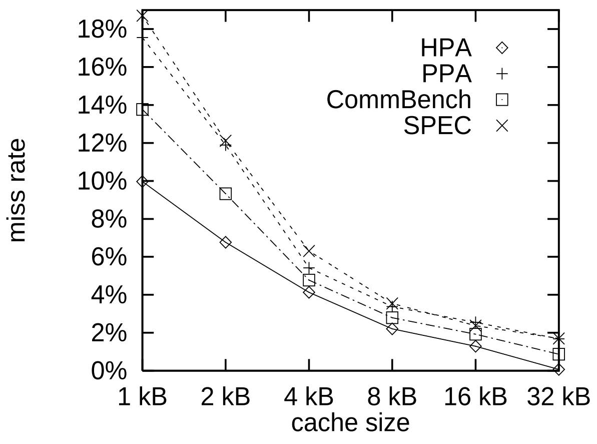


(f) PPA Data Cache (Decoding)

Figure 4: Instruction Cache and Data Cache Miss Rates for *CommBench* Applications



(a) Instruction Cache Average



(b) Data Cache Average

Figure 5: Average Instruction Cache and Data Cache Miss Rates for CommBench, HPA, PPA, and SPEC.

The *CommBench,* and HPA and PPA component cache performance is compared to SPEC in Figure 5. The following points are relevant:

- If a 8 kbyte instruction cache is available, instruction miss rates under 0.5% can be achieved for all but the CAST program. For 16 kbyte instruction caches, all applications achieve miss rates are below 0.2%.

- Due to the relatively small *CommBench* program kernels, *CommBench* instruction miss rates are considerably smaller then SPEC miss rates (3.8% vs. 8.3% for 1 kbyte, 0.6% vs. 2.2% for 8 kbyte, 0.1% vs. 0.5% for 32 kbyte).

- For Payload Processing Applications, encoding and decoding programs have almost identical instruction and data cache performance. Only ZIP has a much higher data cache miss rate for the encoding application.

- Data cache miss rates are below 1% for a 16 kbyte cache, except for the ZIP and FRAG applications.

- Payload Processing Applications and SPEC have a similar data cache performance. The miss rates for Header Processing Applications however is roughly half the miss rate of SPEC. This is due to the fact that the payloads are streamed through the system, touched a few times for processing, and then sent out. Thus there is little data

Cache Type	16 byte vs. 32 byte	32 byte vs. 64 byte
Instruction	1.67	1.18
Data	1.36	1.26

Table 6: Effect of Line Size on Cache Miss Rates. Table entries indicate the ratio of the average miss rate at the lower line size to the average miss rate at the larger line size.

locality. Headers, on the other hand are typically held in memory for a longer time (e.g., the length of the packet time) and require more complex processing (e.g., routing and forwarding).

The instruction miss rates for *CommBench* varies as a function of the cache line size. Table 6 shows that average miss rates decrease with increasing line size. The step from 16 to 32 bytes decreases the miss rates by 1.67 for instruction cache and 1.36 for data cache. Going to 64 bytes decreases the miss rate further, but only by 1.18 for instruction cache and 1.26 for data cache. There are two applications (CAST and RTR) that have higher miss rates with a 64 byte line size than with a 32 byte line size. The overall performance effect of using increased line size depends not only on the miss rate, but also on the miss penalty. This is processor implementation dependent, though, and is therefore not considered here.

5 Design Implications - Examples

To illustrate how *CommBench* can be used in network processor design three examples are considered. First the computational complexity measure is used to estimate processing requirements. Second, instruction set information is considered to see if creation of special purpose instructions might improve performance. Finally, cache miss rate data is used to determine memory bandwidth requirements of a multi-processor ASIC.

5.1 Computational Complexity

One fundamental system design issue concerns estimating the computational power required for a certain data traffic mix. As an example, consider the requirements associated with processing a mix requiring RTR and DRR (for IP header processing), and CAST (for payload encryption).

Using the results from section 4.2 consider the processing requirements of RTR and DRR with a link bit rate of $R_{lnk} = 1.2\,Gbit/sec$. Assuming packet sizes, l, of 576 bytes, the number

instruction pairs	avg. occurrence	max. occurrence
ADD-SUBCC	3.55%	11.6%
LD-SUBCC	3.03%	13.0%
LD-LD	2.36%	20.2%
ADD-LDUB	2.07%	4.84%
SLL-LD	2.05%	6.82%
STB-ADD	2.05%	4.65%
LD-ADD	1.85%	5.35%
ADD-ADD	1.84%	5.85%

Table 7: Most Frequent Instruction Pairs for *CommBench.*

of instructions per second, M, that have to be executed is:

$$M = (N_{RTR,576} + N_{DRR,576}) \cdot R_{lnk} = (0.2 + 0.5)\,\frac{instr}{byte} \cdot 150 \cdot 10^6 \frac{bytes}{sec} = 105\,MIPS.$$

For on-the-fly CAST encryption (also at link bit rate of $R_{lnk} = 1.2\,Gbit/sec$), the header processing overhead for the data stream can be ignored since payload processing dominates the computational complexity. This is generally true for stream data where typically large packets are used. The computational requirements M for this example are:

$$M = N_{CAST,\infty} \cdot R_{lnk} = 104\,\frac{instr}{byte} \cdot 150 \cdot 10^6 \frac{bytes}{sec} = 15,600\,MIPS.$$

This first cut analysis indicates that standard RISC processor by itself is sufficient for header processing at link speed, but will not provide adequate computational power for tasks that perform payload processing. The use of vector processing techniques for streaming applications (i.e., embedded vector processors) [20] or multiple parallel superscalar or VLIW processors on a chip [31] are promising approaches to achieve link-speed payload processing.

5.2 Instruction Set Design

Opportunities are now becoming available for incorporating special nonstandard instructions into processor designs. Such instructions can be incorporated either prior to fabrication or (with some restrictions) dynamically during execution. They can be identified by the programmer who is familiar with the applications or by a dynamic instruction profiling analysis.

CommBench applications were profiled and the most common instruction sequences were identified. Table 7 shows the ten most common pairs of instructions. The most frequent instruction pair is an ADD followed by a SUBCC (compare) and this pair constitutes 3.55% of all instruction pairs that occur. Other pairs are less frequent, but the top eight pairs still occur each more than 1.8% of the time. Longer instruction sequences all had substantially lower frequencies of occurrence.

Consider adding a new "load and compare" instruction which has the functionality of the ADD-SUBCC pair. A typical instruction sequence (shown below) indicates how the pair appears in the kernel of the REED encoding program.

```
...
ld    [%i3 + %g4], %g2
subcc %g2, 0xff, %g0
be    0x104c0
add   %g2, %i4, %g2
subcc %g2, 0xfe, %g0
ble,a 0x104b0
sll   %g2, 2, %g2
add   %g2, -0xff, %g2
sra   %g2, 8, %g3
and   %g2, 0xff, %g2
add   %g3, %g2, %g2
subcc %g2, 0xfe, %g0
bg,a  0x10494
add   %g2, -0xff, %g2
...
```

The destination register of the ADD instruction (%g2) is also a source register of the SUBCC instruction. This dependence makes is possible to combine ADD and SUBCC in a single instruction that performs both an addition and a compare. If such an instruction can be implemented without increasing the cycle time, then each occurrence of ADD-SUBCC can be executed in one cycle instead of two cycles. As a result, the average *CommBench* program would execute 3.5% faster and the REED program (having maximum occurrence of that pair) would execute about 12% faster, a significant amount when dealing with a real time application.

5.3 I/O Requirements for Multi-Processor ASIC

The computational complexity of the *CommBench* applications indicate that a single network processor is insufficient to handle the stream processing requirements of a high-speed data link (see Section 5.1). One approach to increasing computational power is through parallel use of multiple processors. The inherent parallelism of independent data flows makes partitioning the workload onto multiple independent processors relatively straight forward. With current advances in ASIC technology it is possible to implement these processors along with a small amount of cache onto a single chip [7].

For this example assume a system where data packets are received from the link, reassembled, and put into a common main memory. A control processor then notifies one of the multiple network processors to process the packet. The goal of this example is to estimate the average memory bandwidth that is required between the main memory and the multi-processor ASIC.

To a first approximation, the memory bandwidth requirements of a processor depends on its cache size (and design) and the program executing on it. Assume 2-way 8 kbyte instruction cache and 2-way 8 kbyte write-through data cache with a line size of 32 bytes. Assume also that the ASIC network processor is comprised of eight individual processors clocked at 400 MHz. For simplicity say that each of the processors performs one of the eight benchmark applications.

The memory bandwidth *mbw* for an application *a* and cache size *c* is:

$$mbw_{a,c} = (I\,miss\,rate_{a,c} + (D\,miss\,rate_{a,c} * \%\,load_a) + \%\,store_a) * clock * line\,size.$$

For $a = CAST$ and $c = 8k$ the memory bandwidth is

$$mbw_{CAST,8k} = (.0385 + (.0076 * .1985) + .0722) * 400 * 10^6 * 32\frac{bit}{sec} = 1.4Gbit/sec.$$

Using the same expression, the *mbw* for each of the applications can be obtained with the total mbw_{total} being the overall sum.

$$mbw_{total} = mbw_{CAST,8k} + \ldots + mbw_{ZIP,8k} = 8.12Gbit/sec.$$

Assuming a 64-bit wide memory interface, a bus clock rate of at least 130 MHz is required. Naturally, this estimate considers only the average required memory access. A more detailed analysis would have to account for activity bursts and peak bandwidth requirements.

6 Summary and Conclusion

This paper has presented a benchmark, *CommBench,* for use in evaluating and designing telecommunications network processors. Of the eight programs in the benchmark, four are oriented towards packet header processing and four towards payload processing. The benchmark is defined and various characteristics of the benchmark have been presented. Where possible, characteristics of *CommBench* have been contrasted with those of SPEC.

In terms of static code size and kernel instructions, *CommBench* programs are about an order of magnitude smaller than the programs in SPEC . The instruction frequencies for *CommBench* are similar when compared to SPEC across the entire benchmark, but the payload processing applications execute significantly more add/sub, shift, and logic operations.

With respect to the cache performance*, CommBench* applications have only about half the miss rate compared to SPEC. For an instruction cache of 16 kbytes the miss rates drop below

0.1%. For data cache performance, payload processing applications behave like SPEC, and header processing applications have about half the miss rate. This lower cache requirements are due to the smaller kernel sizes associated with *CommBench* applications. Branch and addressing mode statistics, though not presented here, are similar to those obtained with SPEC.

Unlike traditional processor benchmarks, *CommBench* provides clearly defined I/O and computational complexity measures that are directed at the network processor environment. Three examples show how *CommBench* can be used in the initial design process. More detailed processor simulation models may also use *CommBench* instruction trace data during design and evaluation.

References

[1] Adams, C. [1997]. "The CAST-128 Encryption Algorithm," *Request for Comments 2144*, Network Working Group, May 1997.

[2] Choi, H. *et. al.,* [1999]. "Synthesis of Application Specific Instructions for Embedded DSP Software," *IEEE Trans. on Computers*, 48, 6, June 1999, 603-613.

[3] Cisco System, Inc. [1999]. "Cisco 12000 Series Gigabit Switch Routers," *http://www.cisco.com/warp/public/cc/cisco/mkt/servprod/opt/prodlit/gsr_ov.pdf.*

[4] Cmelik, R. [1993]. "SpixTools Introduction and User's Manual," *Technical Report TR-93-6*, Sun Microsystems Laboratories, Palo Alto, CA.

[5] Cmelik, R., Keppel, D. [1994]. "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proc. of SIGMETRICS*, ACM, Nashville, TN, 128-137.

[6] Coffman, K., Odlyzko, A. [1998]. "The Size and Growth Rate of the Internet," *First Monday*, *http://www.firstmonday.dk/issues/issue3_10/coffman/index.html,* 3, 10, October 1998.

[7] C-Port Corporation [1999]. "C-5^{TM} Digital Communications Processor," *http://www.cportcorp.com/solutions/docs/c5brief.pdf*

[8] Curnow, H., Wichmann, B. [1976]. "A Synthetic Benchmark," *The Computer J.*, 19:1.

[9] Edler, J., Hill, M. [1998]. "Dinero IV Trace-Driven Uniprocessor Cache Simulator," *http://www.neci.nj.nec.com/homepages/edler/d4/.*

[10] Gaudiot, J.-L., Lombardi, F. [1999]. "Special Issue on Configurable Computing," *IEEE Trans. on Computers*, 48, 6, June 1999.

[11] IBM Microelectronics Division [1998]. "The PowerPC 405^{TM} Core," *http://www.chips.ibm.com/products/powerpc/cores/405cr_wp.pdf.*

[12] Kane, G., Heinrich, J. [1991]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

[13] Lai, X. [1992]. "On the Design and Security of Block Ciphers," ETH Series in Information Processing, Vol. 1, Hartung-Gorre Verlag, Konstanz, Germany.

[14] Lee, C., Potkonjak, M., Mangione-Smith, W. [1997]. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. Inter. Symp. on Microarchitecture, IEEE Micro*-30, 1997.

[15] McCanne, S., Jacobson, V. [1993]. "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proc. of USENIX Tech. Conf.*, USENIX, San Diego, CA.

[16] MIPS Technologies, Inc. [1998]. "JADE - Embedded MIPS Processor Core," *http://www.mips.com/products/Jade1030.pdf.*

[17] NetBSD Foundation Inc. [1998]. NetBSD 1.3 Release, January 4, 1998.

[18] Rao, T.R.N., and Fujiwara, E. [1989] "Error-Control Coding for Computer Systems," Prentice Hall, Englewood Cliffs, NJ, 1989.

[19] Rivest, R. [1995]. "The RC5 Encryption Algorithm," *Lecture Notes in Computer Science No. 1008*, Springer Verlag, Heidelberg, Germany.

[20] Rixner, S. *et. al.*, [1998]. "A Bandwidth-Efficient Architecture for Media Processing," *Proc. Inter. Symp. on Microacrchitecture, IEEE Micro-31*, 1998.

[21] Sanchez, E. *et. al.,* [1999]. "Static and Dynamic Configurable Systems," *IEEE Trans. on Computers*, 48, 6, June 1999, 556-563.

[22] Shreedhar, M., Varghese, G. [1995]. "Efficient Fair Queuing using Deficit Round Robin," *Proc. of SIGCOMM 95*, ACM, Cambridge, Mass.

[23] Srinivasan, V., Varghese, G. [1998]. "Faster IP Lookups using Controlled Prefix Expansion," *Proc. of SIGMETRICS 98*, ACM, Madison, Wisc.

[24] Standard Performance Evaluation Corporation [1995]. *SPEC CPU95 - Version 1.10*, August 21, 1995.

[25] Tennenhouse, D. *et. al.,* [1997]. "A Survey of Active Network Research," *IEEE Communications.* 35:1, 80-86.

[26] Tensilica, Inc. [1998]. "Application Specific Microprocessor Solutions - Data Sheet for Xtensa V1," *http://www.tensilica.com/datasheet.pdf*

[27] Transaction Processing Performance Council [1998]. "TPC Benchmark C," Revision 3.4.

[28] Triscend Corporation [1999]. "Triscend E5 Configurable System-on-Chip Family," http://www.triscend.com/products/dse5csoc.pdf

[29] Wallace, G. [1991]. "The JPEG Still Picture Compression Standard," *Comm. of the ACM.* 34:4, 30-44.

[30] Weicker, R. [1984]. "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. of the ACM.* 27:10, 1013-1030.

[31] Wolf, T., Turner, J. [1999]. "Design Issues for High Performance Active Routers," Washington University, Tech. Report WUCS 99-19, June 1999.

[32] Woo, S. *et. al.* [1995]. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. of Inter. Symp. on Comp. Architecture*, *ISCA*, June 1995, 24-36.

[33] Ziv, J., Lempel, A. [1977] "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. on Info. Theory.* IT-23, 337-343.