# Evaluating Compiler Support for Complexity-Effective Network Processing *

Pradeep H. Rao and S.K. Nandy
Computer Aided Design Laboratory,
SERC, Indian Institute of Science,
Bangalore 560 012. Karnataka. India.

## ABSTRACT

Statically scheduled processors are known to enable low complexity hardware implementations that lead to reduced design and verification time. However, statically scheduled processors are critically dependent on the compiler to exploit instruction level parallelism and deliver higher performance.

In order to ascertain the suitability of statically scheduled processors for network processing (which constitutes a significant segment of the microprocessor industry), we evaluate the performance of aggressive compiler optimizations for networking applications. We perform our evaluations on two multiple issue processors that extensively exploit static scheduling: in-order superscalar and VLIW. Our results indicate that: (1) the compiler optimizations significantly improve packet throughput on both these processors and this improved throughput is comparable with that reported for agressive out-of-order superscalar processors, (2) the performance advantage of an in-order superscalar processor over a VLIW processor is less than 1.8%, suggesting VLIW processors for complexity-effective designs, and (3) the increase in performance due to doubling processor frequency is limited to a maximum of 37%, due to instruction and memory latencies.

## 1. INTRODUCTION

Increasing growth in networking technology demands higher performance and increased flexibility from the processing element in the networking hardware. This has lead to the introduction of several network processors in the market [13]. Future network processor designs will be expected to deliver higher performance with low energy consumption and minimal costs while being flexible enough to support complex protocols, all within a specified time-to-market. In-order to enhance performance, it is critical to exploit more parallelism and to push clock speeds. Parallelism can be exploited at the instruction level using multiple instruction issue and two key architectures that exploit multiple instruction issue are superscalar and VLIW[1].

Superscalar processors issue varying number of instructions per cycle and are either statically scheduled or dynamically scheduled. Statically scheduled superscalar processors exploit compiler techniques and execute in-order while dynamically scheduled superscalar processors use out-of-order execution. Out-of-order superscalar processors require extensive hardware support to eliminate false dependencies [18] and to maintain precise exceptions [19, 20] which do not scale well with increasing issue width [1]. The hardware complexity associated with dynamic scheduling affects clock speed, increases energy consumption and adversely affects development time and costs.

In-order superscalar architectures only allow in-order issue of instructions. The dependence analysis is performed in the decode stage of the processor pipeline and allows independent instructions to issue during a memory stall. Hence the performance potential for this model is higher than that of the VLIW model (described below). However, the dependence analysis hardware increases implementation complexity which worsens with increasing issue width. But, the overall complexity is much lower compared to that of dynamically scheduled superscalar processors.

VLIW architectures, in contrast, specify multiple instruction issue as an instruction set architecture (ISA) feature. The compiler performs static scheduling and packs independent operations together as a long word instruction. The available parallelism among instructions is explicitly represented by the instruction word. Hardware interlocking is also absent in most VLIW processor designs. Consequently, VLIW processors have simple hardware implementations which allow improved efficiency in terms of power and costs while enabling high frequency processor designs.

As the VLIW instruction format represents the function and the number of resources on a given implementation, the code generated is implementation dependent and requires recompilation for each implementation of the architecture. In cases where recompilation is undesirable, the in-order superscalar model discussed above may be used to retain complexity effectiveness. Thus, we find that statically scheduled processors offload complexity from the hardware onto the

*This work is partially supported by a research grant from STMicroelectronics

[1]Very Long Instruction Word

compiler, making statically scheduled processors a popular platform for the design and development of application specific processors (ASP) [4].

Previous architectural evaluations for network processing have focused mainly on dynamically scheduled architectures [11, 10, 9]. While the benefits of static scheduling for application specific network processors are apparent, there is no literature that reports performance data for this class of processors. This paper aims to provide performance data that could help in the design and development of statically scheduled network processors. In this context we evaluate the performance of aggressive compiler techniques such as the superblock [5] and hyperblock [6] optimizations that exploit microarchitectural support for speculation and predication [7] respectively. We explore these optimizations in the context of two processor architectures: in-order superscalar and VLIW, with varying degrees of support for these optimizations. As we expect future network processors to work at higher clock frequencies to increase performance, we also evaluate the performance of static scheduling on these architectures in the face of increasing clock frequencies.

The rest of the paper is organized as follows: Related work is presented in Section 2. Section 3 presents a brief description of the compiler optimizations and the architectural support required for the optimizations to be effective. In Section 4, we describe our simulation environment and the workload used in the experiments that follow. The experimental results are presented in Section 5, which begins with a study of application characteristics that motivate statically scheduled ASPs for network processing. We then evaluate these architectures along with the compiler optimizations that exploit these architectures. An analysis of the effects of increasing frequency on the performance of these two models is also presented in this section. Section 6 concludes this paper with avenues for future work.

## 2. RELATED WORK

Though many network processors based on different architectural platforms are available commercially, there is little performance data reported in literature. Moreover, the reported studies predominantly focus on dynamically scheduled processor architectures. The performance data for networking applications, across different platforms, play a critical role in the choice of architectures for future network processors.

Crowley *et al* [11] studied the performance of three networking benchmarks on several dynamically scheduled architectures, including an out-of-order speculative superscalar, a fine grain multithreaded processor, a single chip multiprocessor and a simultaneously multithreaded processor. Their results showed that simultaneous multithreading was best suited for networking applications, as it exploits both instruction and thread level parallelism.

Memik *et al* [9] presented Netbench, which consists of networking applications representative of packet processing at different network layers. They evaluated these applications using an out-of-order superscalar simulator and compared its performance with media processing applications. They showed that networking applications differ significantly from media processing applications, necessitating a separate benchmark suite for network processors.

Wolf and Franklin [10] proposed Commbench, for the evaluation and design of telecommunication network processors. Commbench consists of applications representative of packet header and data processing. They profiled and characterized the applications using an instruction profiling tool. Their work also reports cache performance evaluated using an uniprocessor cache simulator.

These early studies reported in literature have predominantly focused on out-of-order superscalar processors. Though these processors are shown to be effective, this focus limits the design space. With evolving avenues for network processors (e.g. embedded systems) imposing additional design specifications such as high performance at low power and reduced time-to-market, it may become necessary to consider alternate architectures. Consequently, in this paper we motivate and evaluate the performance of networking applications on two statically scheduled processors: a VLIW and an in-order superscalar processor.

## 3. COMPILER OPTIMIZATIONS

This section describes the compiler optimizations in brief. The compiler optimizations considered in this paper can be categorized into: (i) classical/basicblock (BB), (ii) superblock (SB) and (iii) hyperblock (HB) optimizations. Classical optimizations [14] perform traditional local optimizations and include constant propagation, copy propagation, constant folding, strength reduction etc.

Superblock optimizations [5] form superblocks, add loop unrolling and compiler controlled speculation, in addition to the basicblock optimizations. A superblock is a structure with a single entry, multiple exits for the control flow. In other words a superblock has no side entrances. Compiler controlled speculation allows greater code motion beyond basic block boundaries, by moving instructions past conditional branches. The correctness of code thus scheduled depends on the speculation model assumed and the processor support for speculation. As processor support for speculation leads to complex hardware implementations, we assume the general speculation model [12] in our experiments. The general speculation model enforces lesser restrictions on the instructions that can be speculated without significantly increasing the hardware support required. In this model all potentially excepting instructions have a non-excepting version, and this version is used in the schedule when a potentially excepting instruction is to be speculated. The non-excepting versions for all potentially excepting instructions adds to the additional hardware support.

Hyperblock optimizations add predicated execution (conditional execution/if-conversion) to superblock optimizations. Predicated execution can eliminate all non-loop backward branches from a program. A hyperblock is a set of predicated basicblock in which control may enter from the top, but may have multiple side exits. Hyperblocks are formed using modified if-conversion and are described in detail in [6]. We next present the experimental methodology and detail the parameters and assumptions for each of our architecture models.

# 4. METHODOLOGY

This section presents our experimental methodology and details the parameters for the two architecture models used in simulation.

## 4.1 Simulation Environment

The compiler and performance analysis tools used in this study are based on the IMPACT infrastructure [16]. IMPACT includes an ILP compiler, a trace driven profiler and simulator and a machine description language (MDES) [3]. The compiler provides a choice of three different optimization paths: (i) classical (basicblock) [14], (ii) superblock [5], and (iii) hyperblock [6], along with optimizations associated with these structures.

| Processor Core | |
|---|---|
| Parallelism | n issue |
| Fetch Width | n |
| Fetch Buffer | 3n |
| **Pipeline Stages** | |
| Instruction fetch | 1 |
| Decode/Dispatch | 1 |
| Execute | Table 5 |
| Writeback | 1 |
| **Register Files** | |
| Integer | 64 registers |
| Floating Point | 64 registers |
| **Branch Architecture** | |
| 1024 entry BTB, 2 way associative, 2 cycle misprediction penalty, 1 branch/cycle, saturating 2 bit counter type. | |
| **Memory Subsystem** | |
| L1 I cache | 2KB, 2 way, 64B blocks. |
| L1 D cache | 4KB, 4 way, 64B blocks, 8 combining write buffers. |
| L2 cache | 1MB, direct mapped, blocking, 256B blocks, 4 cycle latency. |
| Memory | 30 cycle latency, Page Size: 4KB, Page Buffer: 4096 pages. |

**Table 1: Base architecture model parameters**

The low level intermediate representation (IR) of the IMPACT compiler – *Lcode*, provides a generic architecture independent representation to evaluate the intrinsic characteristics of network processing workloads. The Lcode representation is essentially a generic instruction set consisting of simple operations. While the Lcode instruction set is similar to those found in RISC architectures, it is not biased toward any particular architecture, making it ideal for architecture independent evaluations.

The simulator enables a cycle accurate trace-driven simulation of a variety of parameterized architectural models (specified by MDES), including VLIW and in-order superscalar architectures. The parameters common to both these architecture models used in our evaluation is illustrated in Table 1. The instruction latencies for the base model is shown in Table 5. Model1 and Model2 are used in modeling high frequency effects and are also indicated in Table 5.

The following section presents a brief description of the workload used in our experiments.

## 4.2 Workload

Our workload is characteristic of applications involved in network processing and are drawn from the following benchmarks available in the public domain: Commbench [10], Netbench [9] and MiBench [8]. Table 2 describes the applications used in this study along with their inputs (provided with the benchmark). The mix of applications chosen for this study are intended to be representative of the processing that occur at the different layers of packet processing. All applications were simulated to completion.

# 5. EXPERIMENTAL RESULTS

Our first set of experiments characterize the network processing applications using IMPACT's IR, Lcode, which enables an architecture independent evaluation of the workload. The results presented in this section motivate a further study of static scheduling.

## 5.1 Application Characteristics

The results in this section characterize network processing applications, spanning a large design space. The IMPACT Lcode profiler was modified to extract the results presented in this section. We begin by examining the opcode frequencies.

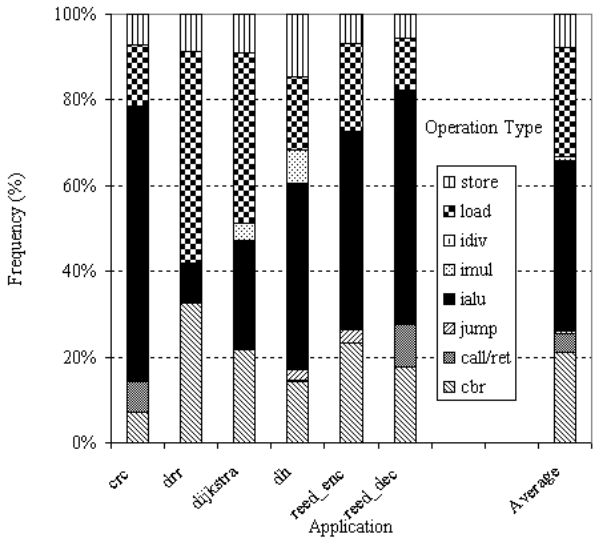


**Figure 1: Operation frequencies for networking applications**

### 5.1.1 Opcode Frequencies

In order to be able to define the balance of resources in a network processor, one needs to consider the operation frequencies. The aggregate results under different operation types is plotted in Figure 1. We find that integer arithmetic operation frequencies are significant and account for an average of over 40% of all operations executed. The incidence of integer arithmetic operations in network processing workloads are 26% higher than that in SPECint92 [15] and is lower than that of media processing applications [2] by about 10%. In order to identify operations that account for the large fraction of integer arithmetic operations, we plot the breakup of integer arithmetic operation frequencies as shown in Figure 2. We find that additions, shifts (operation

| Benchmark | Description | Input Flags | Dynamic Instructions (M) |
|---|---|---|---|
| crc | Calculates a checksum based on a cyclic redundancy check | large.pcm | 372.5 |
| drr | Deficient round robin fair scheduling algorithm | -q 10 {1M packets} | 956.8 |
| dijkstra | Calculates the shortest path between nodes | input.dat | 182 |
| dh | Diffie-Hellman public key encryption-decryption mechanism | number of keys = 10, keylength = 128 | 321.5 |
| reed_enc | Reed-Solomon encoder. (Forward Error Correction) | reed_enc.data | 483.9 |
| reed_dec | Reed-Solomon decoder. (Forward Error Correction) | reed_dec.data | 1,210.8 |

**Table 2: Workload description.**

type:bit) and logic operations account for over 80% of the integer arithmetic operations.

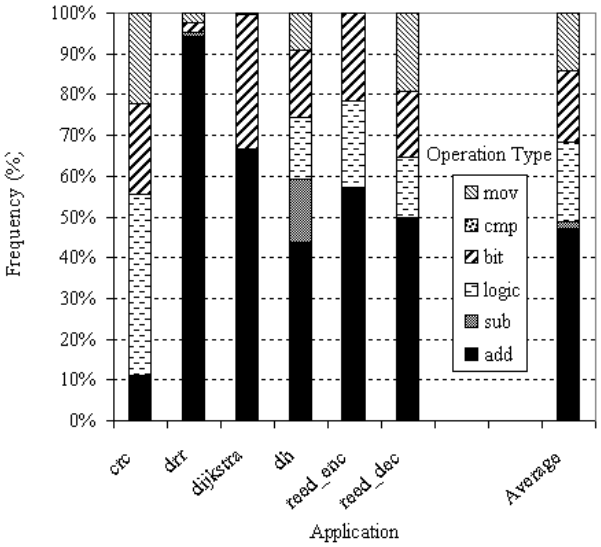


**Figure 2: Operation frequencies for Integer ALU operations**

We also studied the distribution of operations when superblock and hyperblock optimizations are employed. The operation frequencies for the superblock optimization (not shown) do not deviate significantly form that shown for basicblock optimizations. This suggests that superblock optimizations do not cause any additional stress on the resources provided in the processor. Figure 3 shows the operation frequencies when hyperblock optimization is employed. We observe that the hyperblock optimization effectively reduces the number of conditional branches by if-conversion. Consequently, the predicate instructions show up in the distribution and vary between 0% *(crc)* to 37% *(dijkstra)* of all instructions executed.

### 5.1.2 Branch Statistics

Aggressive compiler techniques such as superblock and hyperblock optimizations build and optimize larger structures that consist of several smaller basic blocks taken from the important paths of execution to increase instruction level parallelism (ILP). The optimization of the important paths over the unimportant paths results in increased performance. As these optimization techniques rely on the performance of static branch prediction obtained via profiling, we evaluate static branch prediction performance for network processing applications. The branch prediction accuracy with training inputs is shown in Figure 4 and averages 92.32%. The prediction accuracy for evaluation inputs which differ from those used for training does not exceed 9% for the applications considered. These static prediction rates are higher than that obtained for media processing applications [2]. These high prediction rates improve compiler accuracy and enable superblock and hyperblock optimizations to extract higher performance from support for speculation and predication.

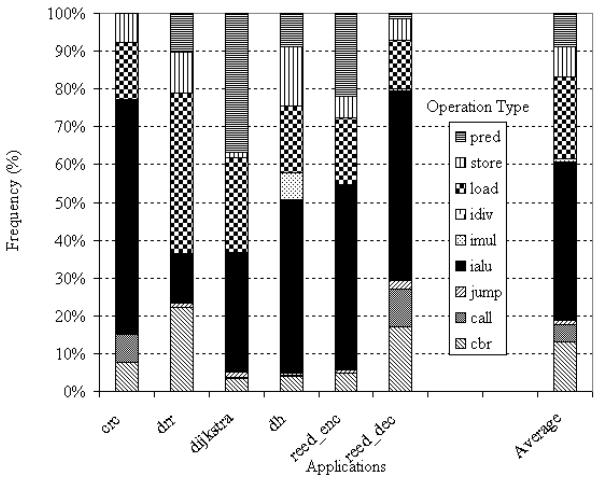


**Figure 3: Operation frequencies with hyperblock optimization.**

Figure 4 also shows the static prediction rate for superblock and hyperblock optimizations. Since superblocks and hyperblocks are formed using profile information, the prediction accuracy for these blocks increase. An exception to this observation is the behaviour of *dijkstra*, for which the prediction rate obtained with hyperblock profiling is less than that obtained with basicblock profiling. We hypothesize that the high degree of if-conversion in the case of *dijkstra* (as seen from Figure 3 reduces the total number of branches significantly while the critical branches that are not strongly biased remain, leading to a drop in the branch prediction rate with hyperblocks.

### 5.1.3 Block Statistics

Figure 5 shows the block size for networking applications for the three optimization paths. The blocksize indicates

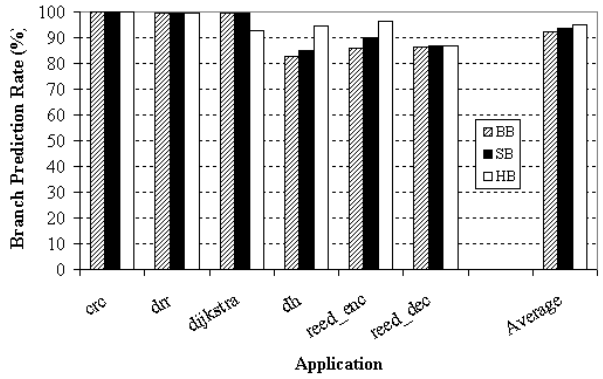


**Figure 4: Static branch prediction performance.**

the number of instructions in a block (basicblock, hyperblock or superblock), and is indicative of the potential parallelism that is available within that block. We observe that, for networking applications, on an average there exists 5 instructions per basic block. The use of agressive compiler optimizations increases the instructions per block to about 13, which is more than twice the original number.

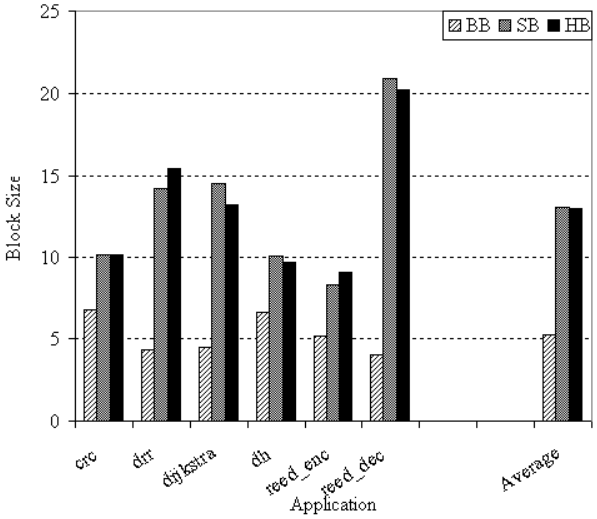


**Figure 5: Average block size for each application, for (i) basicblock, (ii) superblock and (iii) hyperblock.**

### 5.1.4 *Cache Performance*

Though earlier studies have analyzed cache performance for networking applications, we repeat the study in-order to capture the effect of superblock and hyperblock optimizations on cache performance, and also to fix the cache parameters for the processor configuration used in later experiments.

We performed a regression analysis by varying the cache size for the instruction and data cache. The parameters used in this experiment are indicated in Table 1. Figure 6(a) and Figure 6(b) indicate the data and instruction cache performance respectively. Each of these performance plots are obtained for the three optimization paths (classical (BB), superblock(SB) and hyperblock(HB) optimizations).

The average data cache performance is unaffected by the superblock and hyperblock optimizations except for *dijkstra*, which shows a marked increase in the miss rate at lower cache sizes. The instruction cache in contrast is significantly stressed (increased miss rates for equivalent cache sizes) by the compiler optimizations. For equivalent cache sizes, on an average, superblock and hyperblock optimizations result in a 40% increase in cache miss rate over that caused by classical optimizations.

The size of the instruction and data cache was chosen such that the average miss rates are below 1% across all optimization paths. This works out to 1KB for the instruction cache and 4KB for the data cache, across the applications considered. These values are much lower than the equivalent cache sizes required for SPECint92 [10] or Mediabench [2], due to the much smaller program kernels.

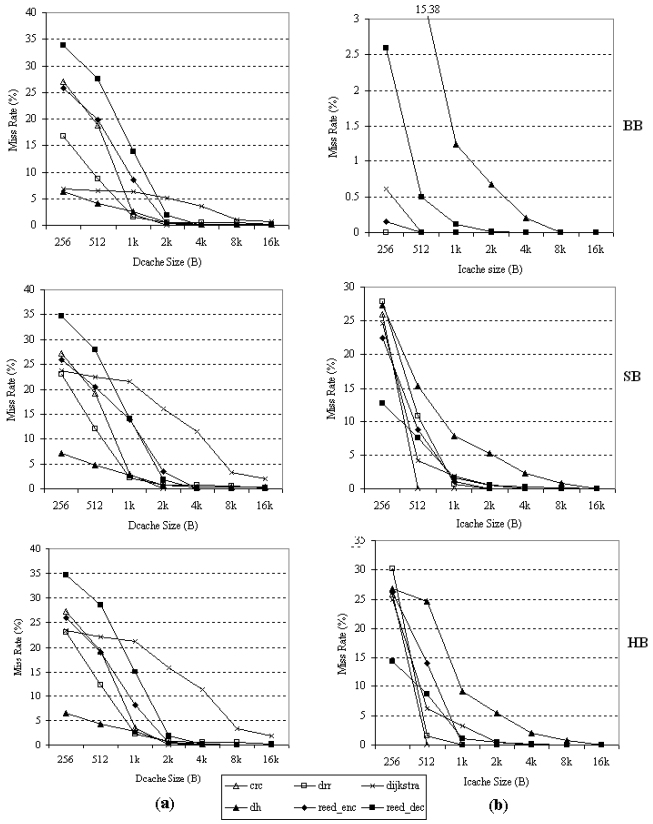


**Figure 6: (a) Data cache and (b) instruction cache performance for basicblock (BB), superblock (SB) and hyperblock (HB) optimizations.**

## 5.2 Architectural Evaluation

The previous section presented results which indicate that static scheduling holds considerable promise for networking applications. We continue our study of static scheduling for networking applications using two processor models that exploit static scheduling: in-order superscalar and VLIW. This section presents the performance results for these two processor models. To avoid misleading results we chose the total execution time as a metric of performance over other metrics as IPC/CPI, as the compiler optimizations (e.g predication)

introduce additional instructions that skew reported results. Execution time as a metric gives a better picture of true performance.

Figure 7 shows the speedup for the VLIW processor model, for varying issue width averaged across the applications under consideration. The three bars for each issue width corresponds to the three optimization paths and the speedup is reported with respect to that of a single issue VLIW processor, employing basicblock optimization. The speedup plot was obtained with the machine parameters specified in Table 1, but using perfect caches. The speedup plots for the in-order superscalar processor do not significantly differ (< 0.1% difference in execution time) from that shown for the VLIW model, due to our assumption of perfect caches. This is attributed to the performance benefit derived from the in-order superscalar processors due to its ability to issue independent instructions on a memory stall.

The results indicate that, while classical optimizations are unable to exploit increasing issue width, superblock and hyperblock optimizations allow code to run upto 2.2 to 2.4 times faster. In other words the speedup for superblock and hyperblock can be upto 2.2 and 2.4, respectively.

The if-conversion performed by hyperblock optimizations result in additional operations, as instructions on either side of the "converted" branch will be replaced by their predicated version and executed. This additional overhead reduces hyperblock performance at lower issue widths and shows up in Figure 7, with the HB bar lagging the SB bar. However, the availability of additional issue slots at higher issue widths (> 4 issue) result in increased performance with hyperblock optimization. Also, at higher issue widths, the increase in performance due to hyperblock optimizations, over superblock optimizations does not exceed 8%. This small improvement in hyperblock performance might not justify the additional hardware required.

From Figure 7, we observe that the performance delivered by all optimizations level off beyond 4 issue, indicating reduced processor utilization. This suggests an ideal issue width of 4 for statically scheduled processors for networking applications. However, the additional issue slots can be exploited to support the considerable thread level parallelism available in networking applications and this study is the focus of our current research.

We also evaluated the effect of cache on the performance of networking applications using the cache configuration (see Table 1) determined earlier. The maximum percentage increase in execution time for the in-order superscalar (IOS) and VLIW models is indicated in Table 3.

| | IOS | VLIW |
|---|---|---|
| Basicblock | 1.06% | 1.08% |
| Superblock | 5.6% | 6.8% |
| Hyperblock | 5.6% | 7.4% |

**Table 3: Maximum increase in execution time due to real caches.**

Table 3 indicates that the effect of real caches are more pronounced on the execution time on a VLIW processor, more so with agressive optimizations and increasing issue widths (Table 4). This is not surprising as the in-order superscalar processor model allows independent instructions to issue on a cache miss whereas the VLIW model stalls.

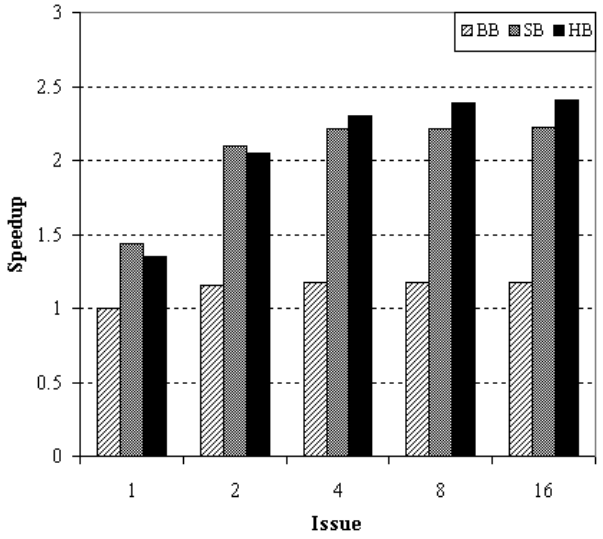


**Figure 7: Speedup with superblock and hyperblock optimization with varying issue.**

| Issue Width | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| In-order Superscalar | 6.41 | 6.48 | 6.25 | 5.66 |
| VLIW | 6.59 | 6.93 | 7.84 | 7.41 |

**Table 4: Maximum increase in execution time due to real caches for different issue widths.**

However even with a realistic cache configuration, the average performance for the most agressive in-order superscalar processor (16 issue with HB optimization) model does not exceed the performance obtained with an equivalent VLIW by more than 1.8% (not shown). This behaviour can be attributed to the nature of networking application kernels. The low cache miss rates diminish the performance advantage obtained due to issue of independent instructions by the in-order superscalar processor.

The lack of sufficient performance advantage with an in-order superscalar processor suggests that the VLIW architecture could be a better platform due to its complexity-effective design (owing to the absence of scoreboarding).

We present a "back-of-the-envelope" calculation to determine network rates. We consider the performance of an 8 issue VLIW processor (with SB optimization) on the benchmark *drr*. Processing 1M packets takes 453.8M cycles on the base model. Assuming a 500MHz clock rate and an average packet size of 750B (packet size uniformly distributed between 40B and 1500B), this works out to a network rate of 6.6Gbps. Doubling the clock frequency to 1GHz (see Section 5.2.1) pushes network rates for *drr* to 13.25 Gbps. As we do not use the same benchmark set used in other studies, we are unable to compare network rates with dynamically

scheduled processor architectures. But, we believe that the numbers shown above are competent for a processor running a single thread of execution. Further research on exploiting thread level parallelism (TLP), on statically scheduled processors could bring about dramatic improvements in network rates, acheived at low hardware complexities.

| | Base Model | Model 1 | Model 2 |
|---|---|---|---|
| Frequency | 500MHz | 1GHz | 2GHz |
| ALU | 1 | 1 | 2 |
| Branches | 1 | 1 | 2 |
| Load | 3 | 5 | 7 |
| Store | 1 | 1 | 2 |
| Multiply | 2 | 4 | 6 |
| Divide | 15 | 25 | 35 |
| Floating-Point | 2 | 4 | 6 |
| L2 cache | 4 | 6 | 8 |
| Memory | 30 | 70 | 150 |

**Table 5: Instruction and memory latencies for three models**

### 5.2.1 *Frequency Effects*

An alternative to increasing processor performance involves increasing the clock frequency. This leads to deeper pipelines and increased instruction and memory latencies, all of which are detrimental to performance. In order to determine the effect of higher clock frequency, we evaluate the performance of networking applications using the latencies specified in Table 5 for the base model (B, 500MHz), model 1 (M1, 1GHz) and model2 (M2, 2GHz). The application code was recompiled and simulated for each of these models.

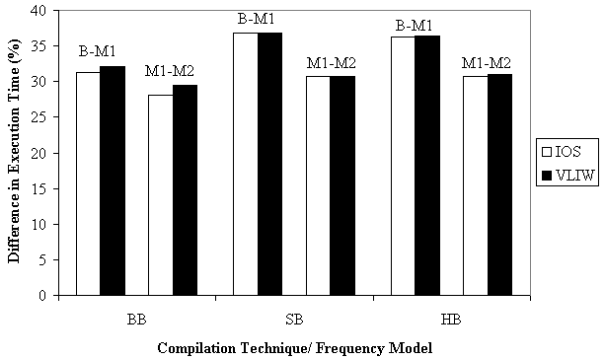


**Figure 8: Difference in execution time in moving from the base-model to model 1 (B-M1) and in moving from model 1 to model 2 (M1-M2). Data shown for the three optimization paths (BB, SB and HB) and two processor models (in-order superscalar (IOS) and VLIW).**

Figure 8 shows that doubling processor frequency (to 1GHz) does not improve performance over 37%. This benefit falls further when moving to 2GHz with the performance benefit being limited to 31%. We also observe that increased frequencies affect both processor models similarly. These results show that the instruction and memory latencies play a significant role in the maximum performance that can be derived from increasing processor clock frequency. These results suggest opportunities for efficient latency hiding techniques (eg. prefetching [17]) to improve performance with increased clock frequencies.

## 6. CONCLUSIONS AND FUTURE WORK

This paper presents the results obtained from a study of static scheduling for networking applications. We summarize the following key results:
(1) The operation frequencies differ from that found in SPECint92 and media applications. This impacts the balance of resources in the network processor.
(2) High static branch prediction performance coupled with large block sizes (after optimization) make static scheduling attractive for networking applications.
(3) Small program kernels result in reduced cache requirements.
(4) The speedup due to superblock and hyperblock optimizations can be as high as 2.2 and 2.4 respectively.
(5) The improvement due to hyperblock optimizations, over superblock optimizations, does not exceed 8%. This small improvement in performance might not justify the additional hardware support for predication.
(6) The performance benefit of an in-order superscalar over a VLIW processor does not exceed 1.8%, making VLIW architectures attractive for designs that require low hardware complexity.
(7) Rough calculations show that statically scheduled network processors *can* deliver competitive network throughputs at low hardware complexity.
We believe that this data could be used as a first step to guide the design and development statically scheduled network processors.

We realize that networking applications offer significant thread level and packet level parallelism that can be exploited to obtain higher packet throughputs. We also note that statically scheduled processors, owing to their simplicity could lead to complexity-effective and high frequency designs. With this view, our current and future research involves investigating techniques to exploit thread and packet level parallelism in statically scheduled processors for networking applications.

We conclude that statically scheduled network processors are relevant, more so in emerging domains such as embedded systems, where the design specifications are stringent.

## Acknowledgements

The authors would like to thank STMicroelectronics for the research grant that partially supported this work, and the anonymous reviewers for their insightful comments on improving and extending this work.

## 7. REFERENCES

[1] Subbarao Palacharla, Norman P. Jouppi and James E. Smith. Complexity-Effective Superscalar Processors. In *24th International Symposium on Computer Architecture*, pp. 206-218, June 1997.

[2] Jason Fritts and Wayne Wolfe. Evaluation of Static and Dynamic Scheduling for Media Processors. In *MICRO-33 MP-DSP2 Workshop*. ACM, Dec 2000.

[3] John C. Gyllenhaal, W.W. Hwu and B. Ramakrishna Rau. HMDES Version 2.0 Specification. *Technical Report*, IMPACT-96-3.

[4] Margarida F. Jacome and Gustavo de Veciana. Design Challenges for New Application-Specific Processors. In *IEEE Design & Test of Computers*. April - June 2000.

[5] W.W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. In *The Journal of Supercomputing*, pp. 224-233, May 1993.

[6] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *27th International Symposium on Microarchitecture*, pp. 217-227, Nov 1994.

[7] M.S. Schlansker, B.R. Rau, S. Mahlke, Vinod Kathail, Richard Johnson, Sadun Anik and Santosh G. Abraham. Acheiving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. *Technical Report HPL-96-120*, HP Laboratories. pp. 224-233, Nov 1994.

[8] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, Dec 2001.

[9] Gokhan Memik, W.H. Mangione-Smith and Wendong Hu. Netbench: A Benchmarking Suite for Network Processors. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pp. 39-42, Nov 2002.

[10] Tilman Wolf and Mark Franklin. Commbench - A Telecommunications Benchmark for Network Processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000.

[11] Patrick Crowley, Marc Fiuczynski, Jean-Loup Baer and Brian Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the International Conference on Supercomputing*, May 2000.

[12] R.A. Bringmann, S.A. Mahlke and WW Hwu. A Study of the Effects of Compiler-Controlled Speculation on Instruction and Data Caches. In *Proceeding of the 28th Annual International Conference on System Sciences*, Jan 1995.

[13] Niraj Shah. Understanding Network Processors. Masters Thesis, University of California, Berkeley. 2001.

[14] A.V. Aho, R. Sethi, J.D. Ullman. Compilers: Principles, Techniques and Tools. *Pearson Education Pte. Ltd.*, 2001.

[15] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach (Third Edition). *Morgan Kaufmann Publishers Inc.*, 2003.

[16] The IMPACT Project. *www.crhc.uiuc.edu/IMPACT*.

[17] Zhigang Hu, Margaret Martonosi and Stefanos Kaxiras. TCP: Tag Correlating Prefetchers. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Feb 2003.

[18] Thomas M. Conte. Superscalar and VLIW Processors. In *Parallel & Distributed Computing Handbook*, chapter 21, edited by Albert Y. H. Zomaya. McGraw-Hill, 1996.

[19] Weiss S. and J. E. Smith. Instruction Issue Logic for Pipelined Supercomputers. IEEE Transactions on Computers, vol c-33, pp 1013-1022, 1984.

[20] W. W. Hwu and Y. N. Patt. Checkpoint Repair for High-Performance Out-of-Order Execution Machines. IEEE Transactions on Computers, vol c-36, pp 1496-1514, 1987.