Microprocessor Optimization for Network Protocol Implementations

Kevin Camera and Tim Tuan
{kcamera, timt}@eecs.berkeley.edu

Computer Science 252, Fall 1999
University of California, Berkeley
Prof. John Kubiatowicz

Abstract

A popular trend in modern computing is the migration from desktop computing to a more distributed environment where individual devices communicate with each other over a network. Such devices, especially those in the wireless domain, are typically limited in resources, most importantly size and power. It is therefore important to be able to implement network protocols efficiently, resulting in high performance with low cost and low power consumption.

For our CS252 course project, we have investigated ways to optimize a modern microprocessor architecture for the implementation of the Intercom wireless communication protocol. The Xtensa[1] processor generation tools were used to configure a sample processor and then tune the architecture optimally for the protocol application. In this study, we have looked only at the transport layer of the protocol, but the results should be valuable to protocol implementations in general. In this paper, we present both the methodology of our analysis and the performance results for our proposed improvements.

1 Introduction

A popular trend in modern computing today is the migration from monolithic, power-hungry desktop computers towards more ubiquitous computing devices that communicate with each other through a local network. An increasing number of these devices are mobile and handheld, communicating over a wireless medium. It is often crucial for these devices to be small, low-power, and inexpensive, yet they need to access the network quickly and frequently. Consequently, these devices require high-performance protocol processing at low cost and low power consumption.

Until recently, network protocols have generally been implemented on high-performance, general-purpose microprocessors. While this approach has been sufficient for traditional needs, the emergence of low-cost computing devices has resulted in demand for greater efficiency in processing network protocols. The question we face is: how can we implement network protocols more efficiently? There are a number of possible solutions to this question, including full-custom hardware, embedded microcontrollers, and low-power microprocessors. Unfortunately, no one thus far can offer a perfect answer.

In our project, we have explored one of the possible solutions, a general-purpose low-power microprocessor. Although protocols have been conventionally implemented in microprocessors, little attention has been given to the efficiency of the protocol operations. Our hypothesis was that protocols in general share common behavior, such as finite-state-machine-like control flow. Our goal was to first find ways to exploit this behavior, and ultimately propose an architecture that is optimized for processing protocols.

To exploit potential optimizations, we have used a novel approach by running a benchmark protocol on a microprocessor featuring an extensible architecture and a full-featured simulation environment. As for the sample protocol, we have chosen the transport layer of the Intercom protocol, a radio-based wireless communication protocol developed at the Berkeley Wireless Research Center. In Intercom, we have a

[1] The Xtensa processor generation tools are products of Tensilica, Inc.

well-defined protocol implementation that is also of realistic complexity and processing load. We took care in our analysis not to look into behaviors that we believe to be characteristics specific to Intercom.

In our analysis, we found several potential optimizations. They are indeed interesting behaviors, but certainly not groundbreaking. Our results indicated a significant amount of execution time spent in memory management routines, and many instructions flushed by taken branches. The protocol application also has an unusually large number of short function calls. The quantitative details of our findings are described later in this paper.

The road we took was not without obstructions. We encountered a number of unexpected difficulties, almost all related to deciphering our design tools. Given better tool support and more time, there is certainly more work that can be done along this line of research. We will summarize the logistic problems we faced and outline any additional work we believe would be valuable should one decide to probe further into this topic.

The following section will describe our methodology, which includes descriptions of the simulation platform and the example protocol. Section 3 presents our observations and results in a qualitative fashion. Section 4 will describe in detail our proposed optimizations for protocol implementation. Section 5 presents the performance improvements yielded by these optimizations. Section 6 addresses possible future work in this area.

2 Methodology

One of the driving motivations for this project is our research methodology. We believe that we have an analysis method that is original and unique, as it provides a new level of flexibility and user control over the processor's specifications and the simulation environment. This is achieved by using state-of-the-art technology in ASIC design provided by Tensilica, Inc. We used their configurable processor generation platform to implement the Intercom transport layer on various processors with different core architecture parameters. The contrasting results of these different processors shed light on features that required improvement.

This section will provide background knowledge of the Intercom project and the Tensilica tools used to generate processors for simulation. Understanding these two aspects of our project will pave the road to understanding the cogency and limitations of our study.

2.1 Intercom Protocol

The Intercom project is an ongoing endeavor at the Berkeley Wireless Research Center. The goal of this project was to design and implement a set of protocols for multi-node, wireless intercom systems. In the short term, only voice transmission is supported. However, one can imagine that the protocol set can be extended to include a wide variety of data types without significant change.

2.1.1 Current Implementation

The current implementation of Intercom consists of four layers: physical, link, medium access control (MAC), and transport. The physical layer uses radio transceivers to establish the physical interface for transmission and reception of voice data. The link layer is responsible for tasks such as error detection and line-balancing. It is implemented entirely in a Xilinx field programmable gate array (FPGA) chip. The MAC layer implements a standard time-division multiple access (TDMA) protocol. The functionality of the MAC layer is split between an ARM processor and the same FPGA. Lastly, the transport layer is responsible for management of the communication channels, performing tasks such as establishing logical connections between two nodes. In Intercom, the transport layer resides solely in the ARM processor.

For our analysis of protocol behavior, we will limit ourselves to the transport layer, as it is the most likely candidate to be implemented purely on a microprocessor. Ideally, we hoped to find any possible efficiency improvements that could be applied to the existing ARM device.

2.1.2 Test sequence

To effectively analyze the runtime performance of the Intercom transport layer protocol, it is important to generate a quality input sequence. We define a high-quality sequence as one that can accurately simulate the performance of the system in a practical application. For our purposes, it is also necessary to make the processor do enough "useful work" to outweigh the overhead potentially caused by starting up the process. With these restrictions in mind, we generated a test sequence composed of incoming signals listed in the table below.

Table 1: Input signal distribution

Signal Description	*Occurrences*
Set node type	1
Enable device	1
Request connection	7
Release connection	7
Query status	2
Synchronize (send SSDB)	2

The test sequence essentially simulates the actions of a home node in the Intercom system. The input signals generate a number of requests to establish and terminate connections. Included are also requests to download the slot-set database (SSDB) from the basestation, which typically occur less frequently.

2.2 Tensilica Processor Platform

For all our simulations, we used the Xtensa processor design environment from Tensilica, Inc. There are several components of Xtensa that combine to provide a complete computing system which is customizable at almost all levels. Xtensa generates a custom processor based on many user-customizable parameters. It also builds a GCC-compatible compiler suite and a cycle-accurate instruction simulator tuned to the attributes of the given architecture. The entire generation process takes approximately half an hour in our experience. The flexibility and rapid generation of processors is crucial to the methodology of our analysis, and made it possible for us to quickly verify the effects of our improvements. Below we present a brief discussion of the tool, starting with hardware synthesis and working up to software development.

2.2.1 Processor Generation

The first step of the design process is to use the Xtensa Processor Generator to specify the core parameters of the processor. The tool asks the user to specify the core frequency, process technology, and general goals of the design, such as whether the generator should make decisions optimized for power, area, or speed. What follows is a long list of customizable parameters, including the number and priority of interrupts, data and address bus widths, total cache and block size, address space boundaries, and scan register debugging support. Once all the appropriate parameters are set, Xtensa will begin generating the processor and the hardware-dependent applications of the software development toolkit. Although it was not relevant in this study, real hardware descriptions ported for several IC design tools are also generated, allowing a direct transition into an embedded ASIC via an existing design flow.

2.2.2 Software Development

Xtensa also provides all the necessary tools to develop the software for an embedded system. During the processor generation stage, a GCC-compliant compiler suite is created specifically tuned the custom architecture. Most of the standard C library routines are also provided in precompiled, statically-linked form. The compiler generates real binary machine code for the new processor, which can be used either in the final embedded system or on the real-time simulator.

2.2.3 Real-time Simulation

The last stage of the Xtensa design process is system simulation and verification. Along with the custom compiler, a cycle-accurate real-time simulator is provided which models the performance of the new processor. The simulator accounts for and displays all delays, including cache misses and block replacement, branch delay slots, and load dependency hazards. This Instruction Set Simulator (ISS), along with a custom version of the standard *gprof* profiling tool, was the foundation of our quantitative analysis throughout this paper.

3 Base System

The first step towards finding areas in need of optimization is to obtain performance information on current implementations. We did so by generating a base system using an Xtensa-generated processor that we configured to closely mimic the ARM SA-1100 processor, which is the processor used in current prototypes using the Intercom protocol. This processor provides a basic level of performance against which optimized variations can be evaluated for their relative performance improvement.

There are two types of performance information available from the Xtensa simulator. The *xt-run* emulation program, a front-end for the ISS, provides a runtime processor performance report. This report includes architecture-specific data such as cache performance, instruction count, and CPI (by category). From this summary, we can see the real hardware's performance, and potentially locate the need for architectural modifications. The simulator also provides an instruction stream trace, which allowed us to get dynamic information about instruction frequency

that would otherwise be hidden in assembly code alone.

The second type of performance report we have is a functional profile of the program execution produced by *xt-gprof* – an Xtensa version of *gprof*. The information we can extract here is a breakdown of various functions in the program based on the amount of time spent by the processor executing each function. This profile can pinpoint the "hot spots" in the program where the CPU is spending significantly more time compared to other functions.

Looking at a combination of these two performance reports, we can gather a very clear picture of processor performance and code efficiency. In the base system, we located three main areas where large improvements can be realized. These three areas – memory allocation, branch performance, and procedure calling – will be summarized in the remainder of this section. Optimizations for these shortcomings will be introduced in the next section.

3.1 Memory Management

One of the most useful pieces of information in the *xt-gprof* profile is the percentage of overall execution time spent in every function in the program. From this we deduced that a significant percentage of the execution time, as much as 33%, is spent in memory allocation routine *calloc*. There is also an additional 11% of the total execution time spent in *memcpy* alone. Clearly this is an area where we hope to improve.

To explain this phenomenon, we first look to the functional behavior of the Intercom protocol. Recall that at the MAC layer, Intercom implements TDMA. TDMA is achieved by time-dividing a fixed-size frame into N time slots, each representing a period during which two nodes can communicate. One of the responsibilities of the transport layer is to manage the channel assignments. This is done by the base station node (which is what we are simulating) in the form of a slot set database (SSDB) that keeps track of the channel usage by each station. Each time a new connection or disconnection request arrives, the SSDB is dynamically updated to reflect the change. Periodically, the base station broadcasts the latest SSDB to each remote station. Updating and broadcasting the SSDB comprises of much of the base station's workload. These actions also require dynamic allocation of memory. This feature of Intercom, or perhaps of TDMA systems in general, partly explains the dominance of memory allocation routines in the overall running time of the transport layer protocol.

For a second explanation for the high occurrence of memory allocation, we look at the tool used to design the Intercom protocol, Telelogic Tau. Telelogic Tau is a communications system design tool that provides a user interface with flow charts and state machines for defining protocol behavior. The source code generated by the tool consists of the user-defined protocol behavior and a kernel which provides all the runtime functionality required by the user code. The kernel is very flexible and general-purpose so it can support the full range of the design language. Such flexibility and broad support results in a large number of calls to dynamic memory allocation routines.

3.2 Branch Performance

In the runtime processor performance summary, we see a significantly large penalty (21% of all processor cycles) due to branch misses. The reason for such a high penalty is that the Xtensa processor does not implement any branch prediction mechanism – it assumes branches are not taken and effectively stalls two cycles every time a branch is taken. This is true for the ARM processor as well [Mon96]. In our study, the penalty is especially high since 31% more branches are taken vs. not-taken. Certainly, this is another area that can be optimized. Before doing so, it is important to identify this behavior as more than an aberration of our particular example by interpreting it in a context of protocols in general.

Branches represent changes in control flow in the program, which are remarkably prevalent in protocols characterized by state machines and flow charts. Most commonly, they come from loops and conditional statements in the source code. It just so happens that Intercom possesses many instances of the former, but there are also reasons to explain this behavior.

Many of the loops in Intercom are by-products of the TDMA control. Many common tasks performed by the base station require indexing through the database. Examples include initializing the SSDB, searching for a free channel for allocation to a remote station, searching for the appropriate channel to disconnect, etc. The existence of these loops lead to a high number of backward branches being taken (N-1 times out of a loop of N iterations). Since TDMA is a common MAC protocol in communication, this behavior should appear frequently in more general cases as well.

3.3 Procedure Calls

The last burden on performance we observed was an extremely high number of quick procedure calls. In fact, approximately 1/3 of the total execution time was occupied by nothing more than calling pro-

cedures. This is very clearly a source of overhead we would like to reduce.

The Xtensa architecture uses register windows for procedure calls. Of the 32 physical registers available on the processor, 16 are addressable at any time, and 4, 8, 12, or 16 can be used within a procedure. On any windowed procedure call, the processor will shift the register window either 4, 8, or 12 registers based on which call is made. Arguments are passed starting at the 5^{th}, 9^{th}, or 13^{th} registers respectively. When a window shift is attempted beyond the range of available registers, a hardware exception is generated and the oldest window must be stored in memory. Likewise, when a window shift is attempted below the last used register, another hardware exception is generated to fetch the data from memory. Based on data from the ISS, the overhead of window overflow and underflow accounted for 20% of the total execution time.

Another side effect of the windowed procedure calls is that each call takes three instructions: *call*, *entry*, and *retw*. Given the large number of procedure calls in the Intercom protocol application, 14.7% of all instructions executed were one of the three listed above.

This rapid procedure calling behavior is another side effect of the code generated by Telelogic Tau. The general-purpose source code is excessively modular, using many small functions to implement the protocol's behavior. Many of these calls are to the kernel's runtime support functions. Clearly, the ability to easily design protocols in a graphical environment is a desirable trait, so we would like to have the processor support such heavily modular code well.

4 Optimizations

Once we had evaluated the performance of the Intercom protocol code on the base Tensilica processor, we were ready to make some optimizations. In this section, we describe some possible improvements to the basic architecture that help to reduce the overall execution time of our benchmark application. We focused primarily on the same three noticeable performance bottlenecks that were introduced above. The quantitative results of our analysis and our improvements are presented in the next section.

4.1 Memory System Improvements

We faced the most unfortunate setback to our optimization attempts while working on the memory subsystem of the processor. The most obvious starting point was with the memory allocation routine *calloc*, which accounted for 32% of the execution time. However, *calloc* is a standard library function and is a precompiled binary file. We had no ability to modify any instructions related to this routine. The same was true for *memcpy*, which accounted for 11% of the total execution time. We tried long and hard to compile our own source code from the GNU standard library, but failed to get it to successfully compile on the Xtensa platform. The architecture was naturally unrecognized to the self-configuration tool, and it was not feasible to port the code by hand within the time constraints of this project.

As a second effort, we evaluated optimizations for indirect loads, which are defined here as load instructions that retrieve a pointer from memory immediately followed by a load instruction off the value of that pointer. To enhance the performance of indirect loads, we propose an *indirect load table*, which interfaces to the data cache. An indirect load opcode would need to be added so the processor can distinguish between typical loads and indirect loads. On the first indirect load to a given pointer, the request would be made to the data cache and the resulting value (the pointer itself) would then be stored in the indirect load table along with the tag of the pointer's address. On every subsequent indirect load instruction to that pointer, the pointer value in the indirect load table is used to index the cache and retrieve the value immediately. Thus, the two loads required before (which inherently had a RAW hazard) are merged into one instruction. This addition will increase the critical path of the cache stage, but in the current version of the Xtensa processor, the cache is direct mapped. Future versions of the processor generator will allow associativity at the same core speed we are using now. Since the indirect load table exhibits delay equivalent to that of an associative lookup, it can be considered a viable substitute for associativity while maintaining the same critical path.

Another improvement we would make to the architecture is a non-blocking load instruction, or a non-blocking cache in general. Our analysis indicated that there is typically a fair number of instructions between the loading of a value and its use as the source of a second computation. In this case, there is a great advantage to allowing instructions to proceed under a miss and overlap the cache miss penalty. If explicit prefetching performed by the compiler were used, the additional hardware required would be minimal compared to a complete non-blocking cache. Minimizing hardware, and therefore area and power, may be preferable for the application space we are considering.

4.2 Branch Improvements

The problem we face with branches is that in Intercom, there exists an excess of branches taken versus not taken. As reported by the run-time performance report, during the execution the program, there were 17468 branch instructions, 9917 of which were taken. This becomes a problem when the processor does not implement any branch prediction scheme, and simply relies on the assumption that all branches are not taken. Such is the case with both Xtensa and ARM. There are several ways to alleviate this problem.

The quick solution is to implement a simple branch predictor, which would incur additional cost in power and area. The performance of the processor would increase dramatically. Modern branch predictors can consistently achieve accuracy over 95%. Even a simple 2-bit predictor can do significantly better than always assuming branches are not taken. Furthermore, recall that the cause of many of the taken branches is the abundance of loops in the protocol. Loops that iterate many times have very predictable behavior; therefore, a simple branch predictor should be able to perform very well in this program.

Let us consider the cost issue in implementing a branch predictor. Originally, the ARM developers decided against branch prediction in favor of a separate optimization: the branch can be resolved in the issue stage even as the condition codes are being updated in execution stage of a previous instruction. This reduced the branch-taken penalty to one cycle at the expense of additional power[2], which was thought to be sufficient performance without implementing a predictor.

However, our case is a little different. We know our code will have a tendency to take branches more often than not, which suggests that perhaps guessing not-taken is not the best possible route. If we are able to dramatically increase our branch prediction success rate, we would be able to afford to remove the extra logic that was implemented to achieve the one-cycle penalty, without sacrificing overall performance. Thus, implementing a branch predictor may either greatly increase performance (if we keep the original optimization in the ARM), or maintain roughly the same performance, but reduce the chip power consumption.

Alternatively, if we keep the branch target adder, a quick fix that can yield better results may be to guess branch-taken instead. This will immediately reduce branch stall cycles, although not as dramatically as any form of branch predictor. The benefit of this alternative is that it requires almost no change to the current processor.

[2] An additional adder was used to calculate the branch target address. Due to critical path constraint, they had to run the adder on every instruction, even the non-branch instructions.

4.3 Procedure Call Improvements

Reducing the procedure calling overhead resulted in the greatest improvement in overall performance. As mentioned in the previous section, the runtime data collected by the simulator indicated that the large number of procedure calls resulted in many register window overflow and underflow exceptions. We decided to reduce the number of these expensive exceptions by increasing the size of the physical register file. This allows a greater "depth" of procedure calls before causing an exception. As the data in the following section shows, the larger register file almost eliminated the number of window exceptions.

The addition of physical registers clearly will add some area and power to the processor. But the performance improvement is so significant that the reduced execution time should result in a net savings in power consumption. Future hardware-level simulation will be needed to verify this assumption with power measurements.

It would be useful to validate compiler techniques for reducing this procedure calling overhead as well. For instance, forcing the compiler to inline some of the most frequently called functions, or completely merging functions called by only one parent, may offer benefits comparable to the increased register file size, but with zero cost in hardware. However, in a mobile embedded system, which we are targeting in this study, the increased code size caused by using inline functions may be undesirable if non-volatile memory is in short supply. Because of our wariness of the software environment (a side effect of weeks spent trying to compile the Intercom source code) and the time limitations of this project, we did not evaluate any of these software techniques in our discussion of performance.

5 Performance

This section provides quantitative results from our simulations and numerical descriptions of processor performance for our proposed improvements. Limitations of the Xtensa processor extensions prevented us from evaluating any branch or memory access modifications in the simulator. In such cases, we include detailed data and calculations of the expected performance of a processor with the new feature. Unless specified otherwise, all measurements regarding program execution time and the dynamic

instruction sequence were made using the summary output of the Xtensa Instruction Set Simulator, a trace of all instruction fetches, and a hefty set of Perl scripts.

5.1 Memory System Results

The first numbers we present are those for the memory optimizations. Although we were unable to attack any of the memory allocation routines, we did find limited success in improving indirect and non-blocking loads.

Indirect loads accounted for 9% of all the load instructions in the program. With an indirect load table as described in the previous section, every one of the indirect load pairs would be merged into a single load instruction, eliminating the second load from the instruction stream. Here we conservatively assume that the indirect load table will not reduce the miss rate or miss penalty, although minor modifications could be imagined that would slightly reduce both quantities. We also assume that the critical path delay is not affected by the table. This seems counter-intuitive, but the Xtensa processor generator guarantees that the next version of the software will produce chips with associative caches at the same clock frequency (most likely due to synthesis or logical improvements). Since we assume that the overhead of the indirect load table will be equivalent to that of cache associativity, we neglect the delay introduced by the indirect load table in our execution time calculations.

The following table describes our expected reduction in execution time. The indirect load table offered a very minor benefit in performance, and may not be a satisfactory answer, especially in a power-conscious environment. If the possibility of increasing the critical path through the cache is considered, this modification clearly would be a bad idea.

Table 2: Performance improvement resulting from indirect load table

Total cycles in program (before)	175657
Total number of loads executed	17034
Number of indirect load pairs	1426
Approx. cycles removed	1426
Total cycles in program (after)	174231
Net execution time reduction	*0.81%*

The second memory system improvement was the addition of a non-blocking load. Analysis of the dynamic instruction stream revealed that an average of 4.656 instructions come between a load and the use of its value. Given the number of load instructions and the miss rate seen in the program, a non-blocking load would save a total of 5,623 cycles, or 3.2% of the total execution time. The table below diagrams the calculation we used for this result. We speculate that such an improvement in performance will justify the additional area (and power) required for providing a lockup-free data cache.

Table 3: Performance improvement resulting from non-blocking loads

Total cycles in program (before)	175657
Number of loads with RAW	13615
Aggregate load miss rate	8.87%
Average number of instructions between a load and use of its value	4.656
Approx. cycles removed	5623
Total cycles in program (after)	170034
Net execution time reduction	*3.2%*

5.2 Branch Results

The table below summarizes branch performance before the addition of any of our improvements. These numbers (provided by the runtime instruction simulator) will be used later in our calculations.

Table 4: Branch performance before improvements

Total Cycles (TC)	175657
Branch Cycles (BC)	51373
Non-Branch Cycles (NB)	124284
# Branch Instructions (BI)	17468
Branches Taken (BT)	9917
Branches Not Taken (BNT)	7551
Average Branch Penalty (ABP)	3.4

The processor performance improves noticeably with our branch optimizations. As we mentioned earlier, in our Intercom benchmark, branch instructions are taken 31% more than they are not taken. Two optimizations we proposed were (1) to implement a simple (2-bit counter) branch prediction scheme, and (2) to always assume branches are taken.

In the first case, it is difficult to generically estimate the performance of a 2-bit branch prediction scheme. As an approximation, we use a safe estimate of 90%[3] overall accuracy. The improvement over the old processor can be calculated as

$$\% \text{ Improvement} = (TC_{old} - TC_{new}) / TC_{old}$$

[3] 90% is the typical performance of a 2-bit predictor in the examples on pg. 264 in Hennessey & Patterson.

Where

$$TC_{new} = NB + (1 + Mispredict\_Rate \cdot ABP) \cdot BI$$

With a two-bit predictor, the misprediction rate is assumed to be 10%. According to the original performance measurements, there were 9917 branches taken, which required a total of 51373 cycles. That computes to 4.4 cycles per incorrectly guessed branch, or a branch penalty of 3.4 cycles. Using the numbers from the original processor performance, we have

$$TC_{new} = 124284 + (1 + 0.1 \cdot 3.4) \cdot 17468$$
$$= 147691 \text{ cycles}$$

Hence, the percentage of improvement with a two-bit predictor with 90% accuracy is

$$\% \text{ Improvement} = (175657 - 147691) / 175657$$
$$= \mathbf{15.9\%}$$

Implementing a new branch predictor would obviously require significant change to the processor. An alternative solution is to simply always assume branches are taken. In this case,

$$\text{Miss Rate} = BNT / BI$$
$$= 7551 / 17468$$
$$= 43\%$$

Which yields

$$TC_{new} = NB + (1 + \text{Miss Rate} \cdot ABP) \cdot BI$$
$$= 124284 + (1 + (BT/BI) \cdot 3.4) \cdot 17468$$
$$= 167290 \text{ cycles}$$

$$\% \text{ Improvement} = (175657 - 167290) / 175657$$
$$= \mathbf{4.8\%}$$

Thus, by simply guessing branches are taken, which requires minimal change to the processor, we gain an improvement of 4.8%. The chart below graphically depicts the improvements from our two potential branch optimizations.

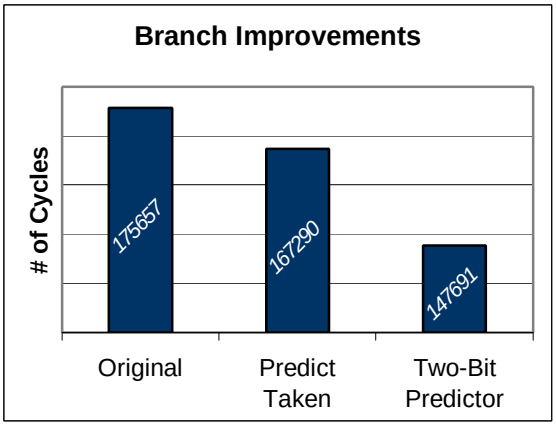

Figure 1: Processor performance improves dramatically with branch prediction. Even an always-guess-taken policy yields some improvement.

5.3 Procedure Call Results

The last numbers, as well as the most encouraging, are those for the increased register file size. The chart below displays the processor performance for the entire program with the original register file of 32 physical registers and the larger 64-register file. As you can see, there is a dramatic 20% reduction in total execution time. The number of window underflow and overflow exceptions almost disappears, dropping from 1267 to only 48. Since all the procedure calls in our benchmark program use an 8-regsiter window, this data implies that the call stack is very rarely any deeper than 8 functions. Xtensa currently only allows the two register file sizes we used here, but a more continuous plot of register file size versus execution time would help a designer size the register file for an optimal balance between power and performance.

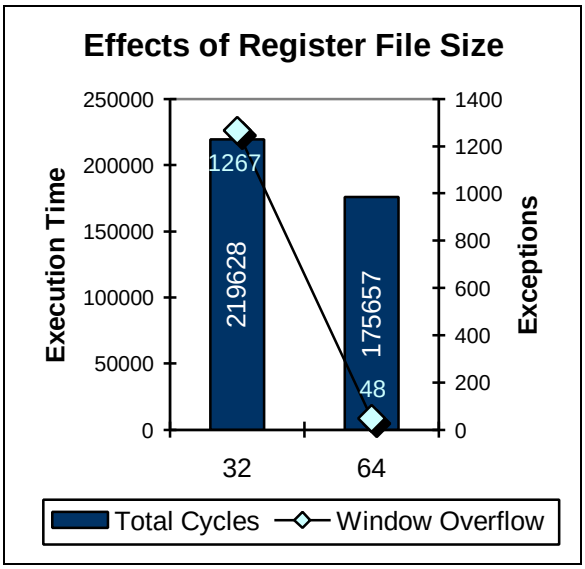

Figure 2: Processor execution time and window overflow exceptions versus register file size.

6 Future Work

This project has served as a initial attempt to find optimizations for network protocols in microprocessor architectures. While we have already found some interesting improvements, we have also discovered that there is much more interesting research to be done in this area. Here, we summarize some of the immediately relevant research that should follow-up the work we performed in this project.

The memory routines were the primary bottleneck in our processor. It would be meaningful to optimize the memory routines by compiling the program along with custom implementations of *calloc* and *memcpy*. By compiling our own source code, we can look for specialized instructions or architectural support for reducing the time required by these functions. Amdahl's Law implies that any optimizations to these routines should show up as a significant gain in overall processor performance.

Another area we fell short on was implementing actual instruction set extensions to verify our improvements. Currently, Tensilica provides minimal support in its instruction extension tool, called TIE. It uses a Verilog-like description language for defining new datapath operations, but only allows register-to-register operations. Since we were most interested in improving memory accesses, procedure calls, and branch behavior, TIE offered us no means of adding our improvements to a real processor. We hope this functionality will be added in the near future, so we can run our modifications through the ISS and verify our calculations in the real dynamic environment.

A third area that warrants attention is regarding the memory management policies used by the protocol. While we can attempt to optimize the memory routines, it may also prove fruitful to investigate the source of the problem, which is how and why the protocol kernel manages memory in such a way. For the purpose of a TDMA protocol, dynamically allocated memory using *malloc*-style block assignment most likely is not the most efficient way of granting memory to processes. Perhaps new allocation algorithms can be conceived that are less flexible and robust, but more practical for running a protocol.

Lastly, in many wireless applications, power is a fundamental concern. In this study, we referred to power sparingly, without providing any real answers to the power issues. Further research to minimize power consumption by protocol processors would certainly be worthwhile. The Xtensa tools do include physical hardware descriptions with the processors it generates, which can provide a vehicle for measuring power consumption in a circuit- or transistor-level simulation.

7 Conclusion

We have presented our project for optimizing microprocessor architectures for protocol implementation. There is a growing need in the computing community for efficient protocol processing. This project was meant to be an initial attempt at solving this relatively new problem.

Our methodology was unique, as we utilized a new configurable processor technology to perform extremely accurate calculations in our analyses. Our results were interesting and thought provoking, but not groundbreaking. Nonetheless, it served as a good foundation for further research in this area. At least within the Berkeley Wireless Research Center, no analysis had been made on the microprocessor's performance in existing prototypes.

Using the Intercom protocol as a performance benchmark, we found many areas that need improvement. For some of them, namely memory routines, branch instructions, and procedure calls, we were able to provide optimizations to the processor to obtain significant performance gain. The details of our optimizations and performance calculations have been thoroughly presented in this paper.

We have only scratched the surface of the problem, and more work is needed in this area. However, we have shown there is much to be gained through optimization of processors for protocol implementa-

tions. More research will certainly uncover even more fascinating results.

8 References

- J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach", Morgan-Kaufmann, 1996
- Intel Corp., Intel StrongARM Microprocessor datasheet, 1999
- J. Montanaro *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE J. Solid-State Circuits*, vol. 31, no. 11, Nov. 1996
- Tensilica, Inc., Xtensa Processor Generator online documentation, 1999