Appears in the Proceedings of the 4th Annual Workshop on Optimizations for DSP and Embedded Systems

Evaluation and Optimization of Signal Processing Kernels on the TRIPS Architecture

Kevin Bush Mark Gebhart Eric Wei Natalie Yudin Bertrand Maher
Nicholas Nethercote Doug Burger Stephen W. Keckler

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

Diminishing performance gains in conventional architectures are driving modern architectures to exploit parallelism more effectively. Next-generation architectures hold promise in the Digital Signal Processing (DSP) arena where high performance and power efficiency are equally important. To better identify optimization techniques on these emerging new architectures, we optimized and evaluated a suite of benchmarks representative of high performance DSP applications. Using these benchmarks, this paper analyzes the performance effects of several code optimizations on a next-generation general purpose processor.

1 Introduction

Technology trends signal a paradigm shift towards architectures that better extract parallelism. It remains to be seen whether conventional optimizations will be effective on these next generation architectures. Representative of these next generation systems, the TRIPS processor is designed to extract high levels of concurrency with an innovative ISA and a grid of processing elements.

We have evaluated eight benchmarks from the Polymorphous Computer Architecture (PCA) C Kernel benchmark suite which was developed by MIT Lincoln Laboratories in conjunction with the DARPA PCA program. The goal of the PCA program is to develop next generation architectures for high performance signal processing of which this suite of kernels is representative [2]. The kernels are designed to test various aspects of a system with a mix of memory and computationally bound algorithms. The TRIPS architecture has the ability to perform well on both memory and CPU bound operations. It uses a banked memory configuration that provides high memory bandwidth and a grid of ALUs that provide opportunity for high IPC on arithmetic operations.

The kernels consist of common signal processing tasks such as matrix transpose, constant false alarm rate detection, singular value decomposition, QR factorization, convolution, and finite impulse response filtering. They were compiled with the Scale compiler which generates code to match the block atomic execution model of the TRIPS architecture [7]. The compiler performs classic scalar optimizations as well as inlining, loop unrolling, and TRIPS-specific hyperblock formation. We then hand optimized the resulting TRIPS Intermediate Language (TIL) - a high level assembly language.

The experiments were run on a cycle-accurate simulator which has been verified against a hardware prototype design. The operand routing delay is a component of the total dynamic execution time and was calculated using the tsim_critical tool which determines the critical path of an execution block [8]. The prototype hardware is currently expected to be manufactured in the Spring of 2006. The hand optimizations show areas where the compiler could be extended to produce better optimized code. To provide a metric for comparison with the general purpose TRIPS processor, we compare the results to the Alpha 21264 microprocessor. While the Alpha chip is not an embedded DSP core, it provides a common baseline as an aggressive high performance general

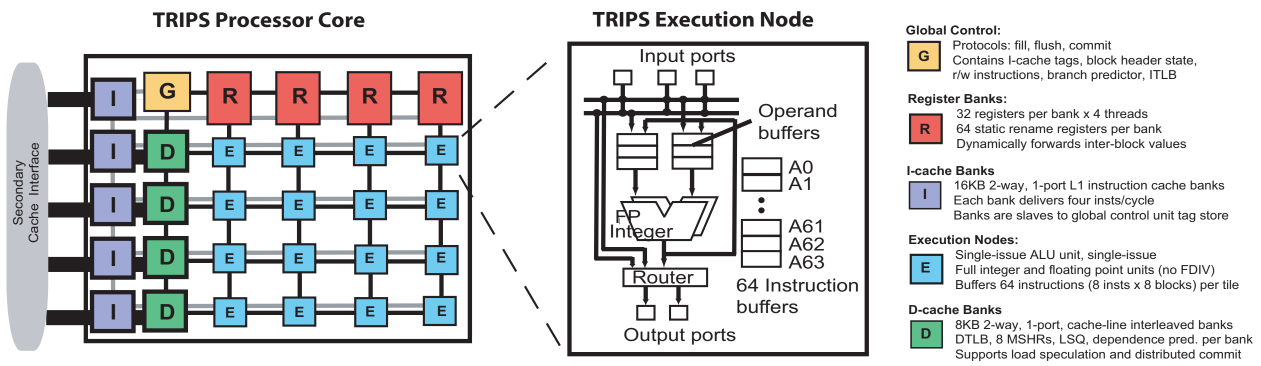
TRIPS Processor Core
Secondary Cache Interface
I
G
R
D
E
TRIPS Execution Node
Input ports
Operand buffers
A0
A1
A61
A62
A63
FP
Integer
Router
64 Instruction buffers
Output ports
Global Control:
Protocols: fill, flush, commit
Contains I-cache tags, block header state, r/w instructions, branch predictor, ITLB
Register Banks:
32 registers per bank x 4 threads
64 static rename registers per bank
Dynamically forwards inter-block values
I-cache Banks
16KB 2-way, 1-port L1 instruction cache banks
Each bank delivers four insts/cycle
Banks are slaves to global control unit tag store
Execution Nodes:
Single-issue ALU unit, single-issue
Full integer and floating point units (no FDIV)
Buffers 64 instructions (8 insts x 8 blocks) per tile
D-cache Banks
8KB 2-way, 1-port, cache-line interleaved banks
DTLB, 8 MSHRs, LSQ, dependence pred. per bank
Supports load speculation and distributed commit

Figure 1: Overview of the TRIPS Design

purpose processor known for its ability to exploit ILP [3, 4].

In this paper, we identified several optimization techniques and evaluated their performance on the PCA Kernel benchmark suite. We found both conventional optimizations such as loop unrolling and architecture specific optimizations such as block merging to be especially effective and complement the unique characteristics of the TRIPS architecture.

The rest of this paper is organized as follows. Section 2 gives an overview of the TRIPS architecture. Section 3 is an evaluation of optimizations applied to each kernel. Section 4 compares the performance of each kernel on TRIPS with the Alpha microprocessor. Section 5 compares the binary size of the kernels before and after optimizations were applied. Section 6 concludes with a discussion of the general optimization techniques along with an analysis of how some optimizations complement others.

2 Architecture Overview

The TRIPS architecture is the first implementation of an Explicit Data Graph Execution (EDGE) ISA, which offers a non-conventional solution to the emerging difficulties of achieving high performance while maintaining power efficiency [1]. The EDGE ISA uses a limited data-flow execution model in which instructions are statically assigned by the compiler to execution tiles and are dynamically executed as soon as their operands are available. By relying on instruction level communication, the architecture removes the need to communicate intermediate results through a global register file and supports distributed out of order execution. Rather than a monolithic processing core, computations are carried out on a grid of replicated ALUs. This allows increased potential to exploit instruction level parallelism. These characteristics give TRIPS the potential to perform well on signal processing algorithms which typically contain a high degree of parallelism.

The TRIPS architecture is designed to exploit ILP while maintaining power efficiency by duplicating general computing resources and removing power hungry structures such as centralized register files. Figure 1 shows a block diagram of the TRIPS architecture. The prototype design contains 16 execution tiles arranged in a 2-D mesh topology. Each execution tile consists of 1 ALU, input ports, operand buffers, 64 instruction buffers, and routing hardware to control operand flow. Execution tiles process instructions concurrently, with a window size of 1024 instructions. The window size is significant because it is an order of magnitude larger than the instruction windows used by conventional architectures. When an operand must be forwarded to multiple consumers on the grid, fanout instructions are used to construct trees that route data. Ideally, instructions are scheduled on neighboring execution tiles to mitigate this routing delay.

Instructions are aggregated into instruction blocks, forming an atomic unit of execution. Blocks are generated and scheduled by the compiler onto the microarchitecture with a limitation of 128 instructions per block. At first glance, these statically scheduled blocks resemble the basic execution unit of VLIW architectures; however the key difference is that instructions are not required to be independent and are dynamically issued [9]. Additional block constraints include a maximum of

32 register reads and writes to the global register banks and a maximum of 32 memory accesses to memory tiles. The TRIPS prototype supports concurrent execution of up to 8 blocks, with 7 blocks executing speculatively.

2.1 Comparison with DSP Cores

At first glance, the TRIPS processor has some resemblance to a DSP core rather than a general purpose processor. However several key differences exist in the ISA, execution model, and specialized memory structures. Two key differences in the current TRIPS ISA from many DSP cores are a lack of special support for SIMD operations and a fused multiply-add instruction. While the current ISA does not support SIMD instructions, support could be added in an ISA extension similar to Altivec. Similar to many DSP cores, the TRIPS processor has many redundant ALUs and a robust means of parallel execution. However, it also provides support for multi-tasking and speculative execution which allows for more general purpose processing.

Many DSP cores have completely separate data and instruction memories often without caches. In the TRIPS design, data and instructions are stored in separate L1 caches; however they are unified in the L2 cache and main memory. Furthermore, the TRIPS L2 cache can be converted dynamically to behave as a local scratchpad memory [10]. This local scratchpad memory is similar to a DSP's notion of a cacheless memory. None of the benchmarks used this feature of the TRIPS architecture but it will be explored in future work. The TRIPS processor is best characterized as a general purpose processor with several features that offer potential in the DSP arena.

2.2 Prototype Simplifications

Several simplifying assumptions were made in order to produce the TRIPS hardware prototype, some of which had a substantial impact on the overall performance of this benchmark suite. Most importantly the hardware lacks support for floating point division, floating point square root, and 32 bit floating point operations. The solution used in the prototype is to emulate division and square root in software and to convert all floating point values to double precision before performing any floating point operations. The cost to emulate the division and square root was between five to ten times higher than a hardware implementation. To guarantee that all bits of the floating point value are correct, the double value must be converted back and forth to a single after each operation.

Kernel	High Level Characterization
QR Factorization	CPU Bound
Convolution	CPU Bound
Finite Impulse Response Filter (FIR)	CPU Bound
Corner Turn (CT)	Memory Bound
Database (DB)	Memory Bound
Constant False Alarm Rate Detection (CFAR)	CPU and Memory Bound
Pattern Matching (PM)	CPU and Memory Bound
Singular Value Decomposition (SVD)	CPU and Memory Bound

Table 1: Overview of the Suite

3 Benchmarks

The PCA suite of benchmarks contains many operations that are representative of signal processing. The represented algorithms are found in the libraries of many DSP applications such as radar, software defined radio, image analysis, and noise filtering. Table 1 shows the eight kernels and their characterizations that comprise the PCA suite. These kernels were chosen to be representative of a wide spectrum of DSP applications with some focusing on memory operations while others stress the system's computational throughput.

Each kernel contains verification code to ensure that the various optimizations applied still preserved the correctness of the algorithm. The base data type for all of the kernels is either integers or doubles. Therefore, the addition of SIMD instructions to the ISA would not benefit the performance unless the implementation was changed to use vectors. In the original implementation, the kernels accept input from data files. In order to avoid clouding the results with file handling, the kernels were modified to accept their inputs from statically linked data sets. The core algorithms that the kernels tested were unchanged.

3.1 CPU-Bound Kernels

In CPU-bound programs, execution time is dominated by computation rather than memory accesses. Several of the signal processing kernels relied heavily on repeated element-wise computations to perform operations such as vector add, multiply, and divide. When computations are independent, the TRIPS grid of processing elements can perform several operations concurrently. When performing vector operations, loop unrolling is crucial to exposing independent loop bodies. In the following sections, we evaluate the QR, convolution, and FIR kernels.

3.1.1 QR

QR factorization is a linear algebra operation that factors a matrix into an orthogonal component Q and a triangular component R. This operation is widely used in adaptive systems and signal processing in conjunction with a triangular solver to approximate over-determined systems, with applications in communication systems, radar, and biomedical engineering. The Fast-Givens algorithm is used, which is characterized by iterations through several loops on disjoint paths composed of fine-grained computations on floating point numbers representing complex data.

The overall goals when hand optimizing were to unroll loop bodies and limit critical paths while satisfying block constraints. We focused our optimizations on instruction reduction followed by loop unrolling. Reducing the number of instructions in each block increased flexibility to unroll loops. To reduce instruction counts, we removed redundant loads, stores, and floating point conversions. The net effect on performance was minor and resulted in slightly shortened execution paths. We attempted to prevent splitting loop bodies across blocks to prevent block overheads. When loop bodies were split across block boundaries the branch from the first block to the second was unconditional, allowing perfect branch prediction. Furthermore, the TRIPS processor was able to begin executing the second block speculatively before the first completed. However, it is still preferable to have single block loop bodies where possible. Loop bodies in QR consisted of 24 floating point instructions and 8 memory accesses, excluding fanout instructions. We were able to unroll by at most a factor of 4 across two blocks. Our analysis also includes unrolling factors of 2 and 3. Table 3.1.1 shows these results.

Version	Cycles	%Speedup
Scale -O4	146,404	-
Unroll by 2	102,865	29.7
Unroll by 3	104,680	28.5
Unroll by 4 (final)	94,528	35.4
FP Conversions Removed	75,519	48.4

Table 2: Results of Optimizations on QR

Due to the limit of 128 instructions per block, the version with an unrolling factor of 3 contains one block with two loop bodies branching to a smaller block containing only one loop body. This undersized block contributed a smaller speedup compared to unrolling factors of 2 and 4. The lack of loop-carried dependences allowed us to duplicate loop bodies without significantly affecting critical paths. Ideally, each loop body can execute independently on the ALU grid, suffering only from operand routing and memory contention. To reduce routing delay, we experimented with various fanout tree configurations, which caused minor performance variations of 1-3%. The construction of fannout trees allows for efficient routing of data along the simple 1-hop network. For our final stage of optimizations, we assumed execution with single precision floating point units and removed all intermediate floating point conversions. Despite producing results with precision errors, this resulted in a shorter critical path and an additional speedup of 13%. This shows that the potential of minor hardware enhancements.

The limitations of the TRIPS grid design on QR are primarily routing delays, which stem from finite computing resources and scheduling. The simplest case of a routing delay is when the consumer for an instruction is located on a separate execution tile. In this situation, the operand must be routed to the execution tile containing the consumer. Each hop between execution tiles suffers a delay penalty of 1 cycle; when instructions are scheduled far apart, the penalty is severe. Routing delays account for 24% of execution time in QR. Despite this limitation, we were able to utilize ALU redundancy and acheived an overall IPC of 3.16.

3.1.2 Convolution

The convolution kernel performs element-wise complex multiplications using a series of filters on an input vec-

Version	Cycles	%Speedup
Scale -O4	313,484	-
Unroll by 3	165,845	47.1
Unroll by 4	126,193	59.7

Table 3: Results of Optimizations on Convolution

tor defined in the frequency domain. The convolution operation is used extensively in DSP, biomedical engineering, and graphics for smoothing, filtering, and image analysis. This kernel uses a loop to select a particular filter and a nested loop to apply the filter to an input vector.

Overall, the basic operations of convolution are similar to QR. The major contrast to QR is that the convolution kernel contained 14 floating point operations and 6 memory accesses per loop body. The smaller loop bodies allowed scheduling flexibility and more aggressive loop unrolling. With smaller loop bodies, we were able to attain a loop unrolling factor of 4 within a block. By confining iterations to a single block, we were able to avoid the overhead of fetching a second block and fanning out its operands to consuming instructions. Our final results appear in Table 3.1.2.

Convolution benefits and suffers from the same architectural characteristics as QR. The computations are aided by ALU redundancy, but also suffer from routing delays. The routing delays in convolution account for 32% of the overall execution time. After all optimizations where applied, we achieved an overall IPC of 5.92.

3.1.3 FIR

The Finite Impulse Response (FIR) kernel is a software implementation of a discrete time filter system. It is commonly used in digital signal processing systems to filter out input frequency components while preserving the phase of the input signal. These characteristics make it extremely useful in applications such as digital communication systems, signal conditioning, audio processing, and radar. The kernel's main operation consists of a base-4 Fast Fourier Transform (FFT), a fast convolution, and a base-4 Inverse Fast Fourier Transform. The FFT and IFFT operations are $O(n \log n)$ and dominated execution time. The optimizations performed on convolution are previously described in Section 3.1.2. The following discusses optimizations on FFT and IFFT.

Version	Cycles	%Speedup
Scale -O4	168,639	-
Instruction reduction	163,269	3.2
Block merging	138,896	17.6
Final	117,437	30.4

Table 4: Results of Optimizations on FIR

Unlike the other kernels which had simple loop bodies, the compiler generated loop bodies for FFT and IFFT that spanned multiple blocks. Subsequently, the potential performance gains from loop unrolling were marginalized. Therefore, we focused on instruction reduction and block merging to reduce the associated block overhead. Our initial optimizations were a combination of general instruction reduction techniques and arithmetic simplification. This included elimination of unnecessary sign extensions, strength reduction, and constant folding. The next series of optimizations focused on minimizing the number of blocks that the inner loop bodies spanned. The number of instructions in the FFT and IFFT inner loops were further minimized by hoisting loop invariant code. Subsequent block merging combined the inner loop bodies of FFT and IFFT into single blocks. Our results are in Table 3.1.3.

FIR suffers from the same performance bottlenecks as the QR and convolution kernels, namely operand routing. Within the inner loops of FFT and IFFT, the percentage of time spent routing operands amounted to 51.3% and 37.5%, respectively. While these routing delays are severe, they are exposed as a high percentage due to the reduced time spent performing parallel arithmetic operations on the replicated ALUs. This is comparable to a conventional architecture in which time is spent performing arithmetic operations instead of operand routing. The final IPCs in the FFT and IFFT inner loops are 5.51 and 4.99, respectively. The overall IPC for the FIR kernel is 2.3.

3.2 Memory-Bound Kernels

Many scientific applications operate on large data sets and therefore the memory capabilities of a system must be considered. Several kernels made extensive access to large structures such as databases and matrices, which placed a heavy demand on the memory system. The TRIPS architecture employs a banked memory system

Version	Cycles	%Speedup
Scale -O4	193,305	-
All Optimizations	58,366	69.8

Table 5: Results of Optimizations on CT

to provide high memory bandwidth. This design allows memory accesses to different banks to be performed concurrently. The following section evaluates the performance of TRIPS on the CT and DB kernels.

3.2.1 CT

The CT kernel performs a matrix transpose on a contiguous block of memory. Matrix transposition is fundamental to linear algebra and is used widely in multimedia, radar, and image analysis applications. By using the corner turn operation, lower dimensional problems can be transformed into higher dimensional problems. Since higher dimensional problems are commonly more parallelizable, this algorithm is commonly used to gather performance gains.

CT contains a nested loop that iterates through an entire matrix swapping the rows and columns. Each inner-loop body contains an address calculation, a load and a store, and no loop-carried dependences. Consequently, when unrolled, each loop body could execute independently. The small size of the inner loop permitted a large unroll factor of 16. Our results are shown in Table 3.2.1.

The TRIPS architecture can achieve a high memory throughput through its banked memory configuration. This kernel was most constrained by the block limitation of 32 memory operations. By exploiting the loop level parallelism and the banked memory system, this kernel achieved an overall IPC of 5.24.

Version	Cycles	%Speedup
Scale -O4	203,273	-
Inlined RBTree	192,690	5.2
Inlined, Unrolled, Merged	184,812	9.1
All Optimizations	164,067	19.3

Table 6: Results of Optimizations on DB

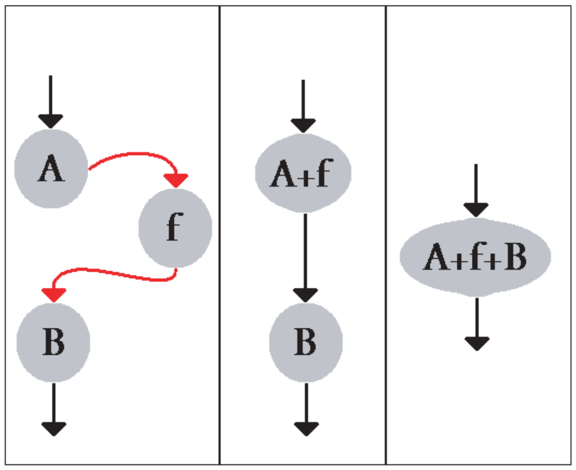

Figure 2: Inlining and Block Merging

3.2.2 DB

The DB kernel operates on a large database of signals. The kernel repeatedly inserts, searches, and deletes various entries in the database. An application of this kernel would be the tracking of a collection of radar signals. The database operations are implemented with a series of red-black tree permutations. Because of the large size of the database, the kernel places a large stress on the memory system. In contrast to CT, after profiling, we discovered that the overall number of loop iterations executed routinely was small. This low loop-iteration characteristic made loop unrolling an ineffective method of optimization.

Nested function calls within the red-black trees commonly used routines made it a prime candidate for inlining. In particular, the cleanup subroutines necessary to re-balance a tree during insert and remove could be deeply inlined. This opened up opportunities to perform block merging and reduced the total number of executed blocks. Block merging is especially beneficial to the TRIPS processor since there is an overhead associated with fetching and executing each block. Since TRIPS blocks are atomic, function calls need to have a separate return block creating artificial block boundaries. These boundaries can be removed in some cases by inlining the function call and merging the return block with the caller block. This subsequent block merging - as shown in Figure 2 - can improve performance significantly by reducing associated overhead instructions and instruction cache pressure.

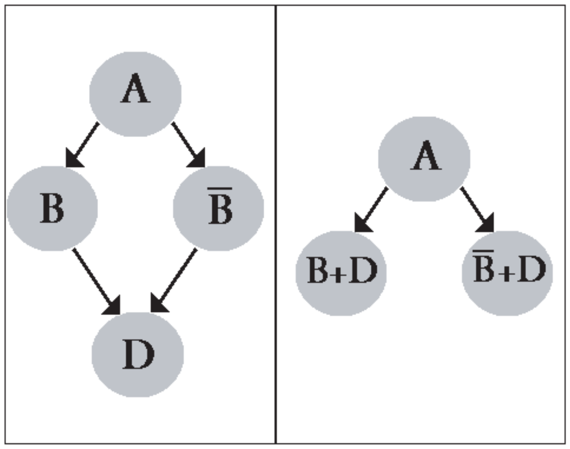

Figure 3: Tail Duplication

While traversing a red-black tree, dynamic control choices often need to be made. These choices were often coded as separate functions. On the TRIPS architecture, both possible execution paths can be included in a single block by using instruction predication. Predicates behave like additional input signals that determine whether or not the instruction will be executed. In this way, we can map significant portions of a control flow graph to a single block, minimizing branches and their associated overhead. The reverse case, tail duplication, also proved beneficial. If the tail block is sufficiently small, it can be predicated and included in its caller blocks as shown in Figure 3. This provides further opportunity to reduce executed block counts and increase block size. Larger blocks offer more opportunity for instruction level parallelism on the TRIPS ALU grid.

We also applied tree height reduction to DB, which was useful in DB's inner function loops. This optimization shortens an expression's overall path length. The TRIPS scheduler is responsible for mapping the reduced-height tree to the ALU grid. By scheduling the newly independent producers adjacent to their consumers, the scheduler can allow the hardware to exploit the concurrency increased by this optimization.

3.3 CPU and Memory-Bound Kernels

Some DSP operations analyze large structures and perform complex operations on data. This places both a computational and memory demand on the system. These algorithms are particularly important to analyze for identifying potential bottlenecks in a systems overall performance on more robust applications. The following sections discuss the CFAR, SVD and PM kernels.

3.3.1 CFAR

The CFAR kernel searches for randomly placed targets in an environment filled with background noise. This algorithm is used in radar, sonar, image processing, and medical engineering. In radar applications this operation is crucial to removing environment noise. The algorithm loops though a data cube and looks for cells with a power exceeding a threshold relative to their neighbors. Instruction merging, block merging and loop unrolling were all found to benefit this algorithm.

The first optimization applied was loop unrolling. After analysis, we found an unroll factor of 4 to be optimal. Similarly to DB, predication was employed to exploit the inherent inner loop parallelism resulting from the absence of any loop carried dependences.

Instructions that produce block outputs such as stores and writes cannot be predicated [6] without nullifying the opposite path. This technique of nullification provides a means of signaling that no output will be generated from the given path. Because CFAR has few block outputs, it is an excellent candidate for nullification when using predication on the parent instruction.

As in DB, function calls created artificial block boundaries that presented opportunities for block merging along disjoint paths. As these blocks were merged, some intermediate values no longer needed to be passed through the register file, removing associated register pressure. Profiling data was used to identify frequently taken paths which could then be given preference to be merged.

The TRIPS architecture was able to achieve an IPC of 2.02 on this benchmark. Choosing the right unrolling factor became one of the critical decisions and required profiling information. Block merging contributed significantly to performance and allowed other optimiza-

Version	Cycles	%Speedup
Scale -O4	190,717	-
All Optimizations	158,342	17.0

Table 7: Results of Optimizations on CFAR

Version	Cycles	%Speedup
Scale -O4	108,317	-
Loop Unrolling	96,535	10.9
Struct Unpacking	88,326	18.5
Block Merging	81,111	25.1
All Optimizations	76,127	30.0

Table 8: Results of Optimizations on SVD

tions such as instruction merging to be applied.

3.3.2 SVD

Singular value decomposition (SVD) is a linear algebra transformation that is commonly used to eliminate noise from data. There are applications for SVD in image processing, seismology, and tomography. In image sharpening the SVD algorithm can be used to discover small singular values which mainly represent noise. There are several different operations with order of magnitude n^3 that make up the SVD operation. These include QR factorization, bi-diagonalization, diagonalization and matrix multiplication. Like the convolution and QR kernels, these operations benefit greatly from the grid of replicated ALUs that allow several different arithmetic expressions to be evaluated concurrently.

Previously discussed optimizations such as block merging and loop unrolling were applied to the SVD kernel. The results are shown in Table 3.3.2. An interesting optimization not discussed previously is struct unpacking. This optimization emerged because SVD uses complex data which is stored as two 32 bit values packed into one 64 bit value. When either of the fields is referenced, both fields must be loaded from memory and unpacked. By storing these values separately, packing and unpacking operations are unnecessary. We found this optimization to have a 7.45% speedup on the SVD kernel.

3.3.3 PM

The Pattern Matching (PM) kernel randomly adds noise to a test signal and compares this signal to a library of test patterns to determine what pattern the signal was originally. This metric for comparison is the weighted mean square error. The combination of the large library and this mathematically intensive algorithm provides a balanced mix between CPU and memory operations. This kernel is representative of the pattern matching needs of many DSP applications including radar and signal identification, where noisy inputs need to be matched to a library of known signals.

The optimizations described in detail in the preceding sections were applied to PM. We found loop unrolling and block merging to be the most effective. Overall, we achieved a 50.3% increase in performance over the compiler generated code. PM's 32-bit floating point operations highlighted the prototypes shortcomings in single precision floating point arithmetic and exposed many prototype-necessary floating point precision conversions. These precision conversions are an artifact of the prototype and artificially increase the length of the critical path. As demonstrated in QR, significant performance gains and further optimization opportunities can be achieved by assuming that 32-bit hardware support is available and removing these conversion instructions.

Kernel Name	Speedup
QR	1.16
CONV	2.21
FIR	1.30
CT	2.06
DB	0.51
CFAR	1.14
PM	0.93
SVD	0.32

Table 9: Speedup Relative to Alpha 21264

4 Comparison to Alpha 21264

In order to make an apples to apples comparison the TRIPS results were compared with the Alpha 21264 general purpose processor. The Alpha 21264 is an industry developed aggressive high performance microprocessor with a mature optimizing compiler. The Alpha 21264 provides an accurate comparison of architectures since its ISA closely resembles that of TRIPS. It has several hardware advantages over the TRIPS prototype such as hardware floating point division and square root functions [5]. The QR and SVD kernels require the square root operation, while QR, FIR, PM, and SVD all require the floating point division operation. All of these benchmarks demonstrated significant gains on the Alpha because of its hardware support for square root and floating point division. In order to generate the Alpha results we used a cycle accurate simulator previously developed [3, 4]. This simulator provides very detailed information about a programs execution and allows fine tuning of parameters to minimize differences in the memory systems of the two processors. The TRIPS processor was able to outperform the Alpha processor on six of the kernels. This highlights the potential of the TRIPS architecture despite prototype shortcomings. The results of the comparison between TRIPS and Alpha are shown in Table 4. Future work would be to make a quantitative comparison of the benchmarks with a DSP core.

5 Code Size

A major concern when applying ILP optimizations such as loop unrolling and inlining is the increase in code size. Since many embedded processors have limited memories the code size metric should be considered when evaluating the effects of different optimizations. Table 5 shows the code size before and after optimizations were applied for the eight kernels. Optimizations such as instruction elimination and block merging reduce the code size. The combination of optimizations that both increase and decrease the code size results in an overall insignificant change in binary size. One obstacle that TRIPS faces with respect to binary size is that nonfull blocks are padded with nop instructions up to either, 32, 64, 96, or 128 instructions. This is similar to VLIW padding where words are padded with nop instructions up to the word size. The only significant change in code size occured in SVD and this can be attributed to a change in the supporting math libraries.

Kernel Name	Initial (bytes)	Optimized (bytes)	Change (%)
QR	433,603	432,731	-0.2
CONV	531,426	531,138	-0.05
FIR	517,251	526,689	1.8
CT	662,287	662,200	-0.01
DB	593,076	593,187	0.02
CFAR	384,740	384,387	-0.09
PM	436,732	437,657	0.2
SVD	662,454	506,571	-23.5

Table 10: Optimization Effects on Code Size

6 Conclusion

Although many optimization techniques were applied, a few stood out as particularly beneficial to the performance of these DSP kernels on the TRIPS processor. In particular, loop unrolling produced performance boosts in kernels such as QR, CT, SVD, and convolution. On the TRIPS architecture, unrolling iterations of loops without loop-carried dependences provided an opportunity to fill blocks and exploit the inherent opportunities for concurrency. While loop unrolling is an effective optimization on conventional processors, it provides additional gains on the TRIPS architecture by exposing more opportunities for parallelism within a block.

Ultimately, the biggest performance gains on the TRIPS architecture can be achieved by reducing the number of executed blocks. Aside from loop unrolling, the number of executed blocks can be reduced by inlining, block merging, and predication. While inlining on a conventional architecture simply eliminates the overhead of a function call, on the TRIPS architecture, it removes an entire block from the execution. Since blocks are large and the overhead of a block is costly, the gains of inlining on the TRIPS architecture are more emphasized. Blocks of sufficiently small size can be arbitrarily merged provided there exists no entry paths to removed blocks. This not only removes a block from execution, but provides opportunities to merge common instructions and expose more ILP. The use of predication works in a similar fashion. By converting a control dependence to a data dependence the number of exposed branches is reduced, thus allowing the grouping of larger segments of code on a single execution block.

Perhaps most notably, we found these optimizations to be mutually beneficial and we performed many of them either together or in succession. Loop unrolling increases opportunities to reduce redundant instructions and perform tree height reduction. Inlining removes the artificial block boundaries created by a function call. Such techniques expose new opportunities for block merging, which in turn provides new opportunities to merge instructions and increase overall ILP.

We were able to achieve an average speedup of 1.83

Kernel	Scale -O4 Cycles	Hand Optimized Cycles	Hand Optimized vs. Scale -O4	Alpha cc Cycles	Hand Optimized vs. Alpha cc
QR	146,404	94,528	1.55	109,652	1.16
Convolution	313,484	126,193	2.48	278,887	2.21
FIR	168,639	117,444	1.43	153,205	1.30
CT	193,305	58,366	3.31	120,234	2.06
DB	203,273	164,067	1.24	83,647	0.51
CFAR	190,717	158,342	1.20	180,510	1.14
PM	258,777	130,227	1.99	121,111	0.93
SVD	108,317	76,127	1.42	24,373	0.32

Table 11: Summary of Results. Average Speedup over Alpha is 1.20, over Scale is 1.83.

over the compiler and an average speedup over alpha of 1.20. While this might seems unimpressive at first glance, the Alpha has a more advanced compiler and hardware advantages such as a floating point square root, hardware divide, and 32-bit floating point operations. Minor hardware improvements in the prototype and migration of optimizations into the compiler could significantly enhance performance. By experimenting with different hand optimization techniques we have highlighted opportunities for additional compiler optimizations on DSP kernels. We have found several characteristics of the TRIPS architecture, such as its limited dataflow and block atomic execution model, to be well suited to provide high performance while still maintaining power efficiency for future signal processing codes.

Acknowledgments

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-4106 and NBCH30390004 and an NSF instrumentation grant EIA-0303609. All opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Finally, the authors would like to extend their appreciation to Ramadass Nagarajan for his many contributions.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *IEEE Computer*, pages 44–55, July 2004.

[2] DARPA. *Polymorphous Computing Architecture Program, http://www.darpa.mil/ipto/programs/pca*, January 2006.

[3] R. Desikan, D. Burger, S. Keckler, and T. M. Austin. Sim-alpha: a validated, execution-driven alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, October 2001.

[4] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *28th International Symposium on Computer Architecture*, July 2001.

[5] R. E. Kessler. The alpha 21264 microprocessor. Technical Report 2, IEEE Micro, March 1999.

[6] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. Trips processor reference manual. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2005.

[7] K. S. McKinly, J. Burrill, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The scale compiler. Technical report, University of Massachusetts, University of Texas, 2005.

[8] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, and S. W. Keckler. Critical path analysis of the trips architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2006.

[9] R. Nagarajan, S. K. Kushwaha, D. Burger, K. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (spdi) scheduling for edge architectures. In *International Conference on Compilation Techniques (PACT)*, September 2004.

[10] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with teh polymorphous trips architecture. In *Proceddings of the 30th Annual International Symposium on Microarchitecture*, pages 422–433, May 2003.