

May 18, 2002. ECE734 Project Report

Mapping DSP Algorithms to General-Purpose Out-of-order Processors

Spring 2002 ECE734 Project Report

Ilhyun Kim and Donghyun Baik

ikim@cae.wisc.edu, dbaik@ece.wisc.edu

Mapping DSP Algorithms to General-Purpose Out-of-order Processors

1.0 Introduction

It is becoming popular to implement DSP algorithms on a general purpose processor. This is not only because many applications often put more emphasis on the cost for developing and maintaining DSP software rather than the hardware cost that may be lowered by mass production, but also because a commodity-part general-purpose microprocessor can easily achieve the performance required for the specific application that might be implemented only by a special-purpose DSP processor. Considering this trend, it becomes more important to find a efficient way to map DSP algorithms onto general-purpose microprocessors.

Many existing DSP transformations try to extract the most parallelism from the algorithm to achieve high performance by mapping multiple independent computations to different processing elements. Similarly, this is also a goal of high-performance compilers that exploit ILP (instruction level parallelism) to hide the latency of operations by enabling overlap of instruction executions. At the same time, more and more general-purpose microprocessors are built with out-of-order execution cores that dynamically re-order instructions so that they keep all available hardware resources running by selecting independent operations within a limited scope over program binary (i.e. instruction window).

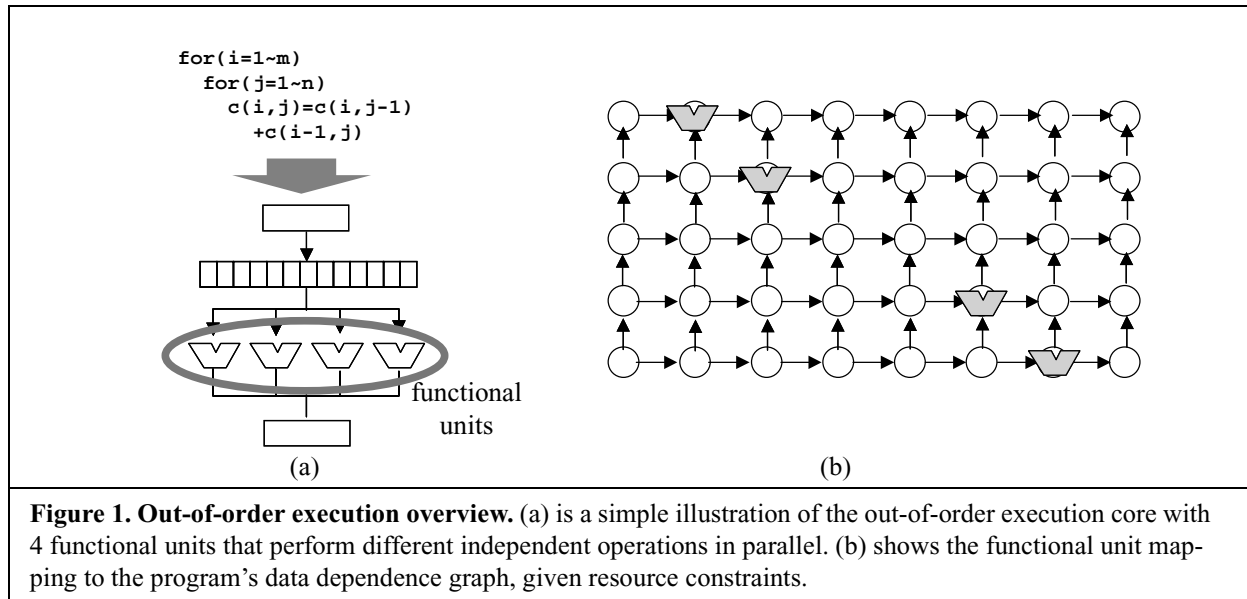
Since these efforts to extract parallelism occur on different layers of high-level algorithm transformations, compiled binary and processor core independently without collaborations, some of efforts may be wasteful and even some of them would negatively affect performance due to the lack of understanding on what actually happens in the microprocessor. In this project, we study the effect of algorithm transformations that may be performed in the high-level program and assembly language levels for DSP applications running on a general purpose out-of-order processor considering several real-world constraint. Specifically, we study the effect of single assignment, unfolding transformations performed on Alpha architecture [2] since they are closely related to optimizing loops that are common in matrix manipulations. Additionally, we are focusing on identifying the source of constraint in loop operations by analyzing data dependence graph.

Based on the findings of efficient algorithm mapping techniques, we present a case study on optimizing MPEG-2 decoder [5] from Mediabench suite. Our optimization shows that the modified decoder significantly outperforms the base code by reducing the execution time by 38%.

The rest of the report is organized as follows: in Section 2, we present the overview on the out-of-order execution core of general-purpose processors. Section 3 describes our simulation methodology and a brief explanations on the Alpha architecture that we modeled. In Section 4, the effect of algorithm transformations running on a general-purpose processor is studied. Finally, Section 5 presents a case study on optimizing a MPEG-2 decoder running on an

Alpha architecture implementation.

2.0 Out-of-order Execution Overview



Many current-generation general purpose processors are built with out-of-order execution core (OoO core) in which the hardware dynamically detects independent instructions with both inputs available and assigns functional units to those instructions [4]. An illustration of the structure is shown in Figure 1a. High-level language is compiled by a compiler that translates the high-level code written by the user into binaries, a machine language primitives that actually realize program semantics. Dynamic pipeline scheduling goes past stalls to find later instructions to execute while waiting for the stall to be resolved. Typically, the pipeline is divided into three major units: an instruction fetch and issue unit, execute unit, and a commit unit. The first unit fetches instructions, decodes them, and sends each instruction to corresponding functional unit of the execute stage. There might be 5 to 10 functional units. Each functional unit has buffers, called reservation stations, that hold the operands and the operation. As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. It is then up to the commit unit to decide when it is safe to put the result into the register file or into memory. The basic model is multiple independent state machines performing instruction execution: one unit fetching and decoding instructions, several functional units performing the operations, and one unit deciding when instructions are complete so that the results can be committed. To make programs behave as if they were running on a simple non-pipelined processor, the instruction fetch and decode unit is required to issue instructions in order, and the commit unit is required to write results to registers and memory in program execution order.

Figure 1b shows the functional units are mapped to the program's data dependence graph, considering the resource constraints. Since the hardware keeps track of data dependences of the instructions, functional units are automatically assigned to possible operations on equi-temporal plane in the data dependence graph if the machine

does not have any resource constraint. However, there are two elements that restrict this functional unit mappings: one is the finite number of functional units and the other is the finite scope over the instructions, which is called instruction window that detects instructions with all data dependences satisfied by the execution of parent instructions. Typically, the instruction window size is 32 to 128 instructions. Even if all data dependences are resolved by the previous executions, the operations should be serialized when independent operations are not within the instruction window since the machine cannot dynamically detect they are ready to issue. However, operations within the instruction window can be issued, trying to keep all resource in the hardware busy to achieve the maximum throughput.

3.0 Simulation Methodology

So far, we briefly explained the out-of-order execution and the limitation of this hardware in terms of searching and executing operations in the program. This section describes a brief explanations on the Alpha architecture that we modeled and our simulation methodology.

3.1 Alpha architecture overview

The alpha architecture is a 64-bit load and store RISC architecture designed with particular emphasis on speed and multiple instruction issue with out-of-order processor core. All registers are 64-bit long and all operations are performed between 64-bit registers. All instructions are 32-bit long. Instructions interact with each other only by one instruction writing to a register or memory location and other instruction reading from that register or memory location. In other words, all array manipulations should access the data only through memory and bring the data into the processor to perform computations.

Figure 2 shows the block diagram of an Alpha 21264 processor. The processor can fetch, decode, execute and commit up to 4 instruction a clock cycle. The processor has a 20-entry integer issue queue associated with the integer execution units and a 15-entry floating-point issue queue associated with the floating-point execution units. For the memory operations, it has 2 general cache ports that read and write data for 2 instruction a clock cycle. The data cache is a 64KB, 2-way set-associated with 64-byte blocks, which is backed up by a 512KB level-2 cache.

Comparing to the dedicated VLSI array structures designed for high-performance DSP applications, we can easily see that the resource and dynamic scope of the 21264 is significantly narrower. In addition, there are other real-world restrictions that degrade the overall performance such as branch mispredictions and cache misses; however, we do not discuss the effect of those restrictions in this report and they are out of the scope of our project.

3.2 Processor model

In order to simulate the dynamic mapping of DSP algorithms onto the processor, our execution-driven simulator is built on SimpleScalar suite [1]. Specifically, we modeled our simulator after the Alpha 21264 architecture. The processor model can fetch, decode, execute and commit up to 4 instructions per a single clock cycle. We also implemented wider machine models with 8, 16, 64, 1024, 4096 fetch/decode/issue/execute/commit bandwidth to examine the effect of resource constraint on the performance given a compiled binary from DSP algorithms. Using

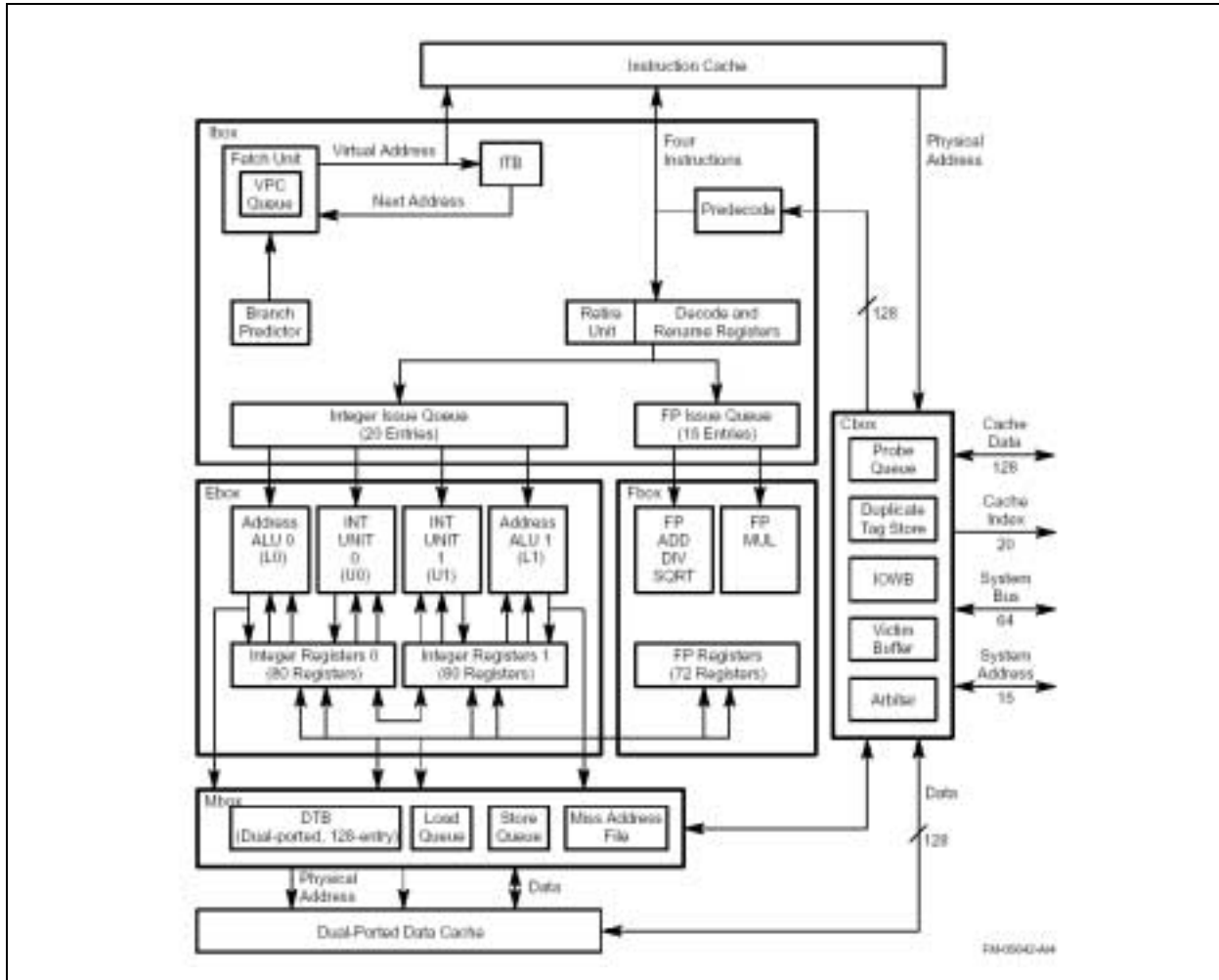


Figure 2. Alpha 21264 processor block diagram.

these models, we can measure the exact execution time count without measurement errors that may occur in real hardware due to other processes running simultaneously and operating system code.

Additionally, we also built an infinite machine simulator that only keeps track of real computations specified in the algorithm itself, ignoring finite machine bandwidth, memory communications and other loop control-related operations. In order to detect only real computations only, we used a simple scheme: the operations related to the real computations in a high-level language e.g. C, is written with floating-point variables and operations while other operations that do not directly count toward the real computations are written with integer variables and operations. We can easily detect the meaningful computations in the simulator to see if they are floating-point instructions. This simulator enables us to figure out any disturbance that may incurred by the compiler that potentially changes the algorithm. In the later sections, we will present that the compiler and its optimization do not alter the algorithm itself by showing the theoretical execution time measured by this simulator.

The compiled binaries are also analyzed by our disassembler tool based on the sim-profile, also derived from SimpleScaler suite. The assembly-level analysis and instruction transformations that exploit the characteristics

of the real hardware will be also presented in the later sections.

3.3 Compilers and Benchmarks

All test code and benchmark programs are compiled with gcc 3.03 running on a DEC alpha processor. We tested the effect of compiler optimizations with various optimization options from O0 (no optimization) to O3 (full optimization) and found that even the full optimization does not change the semantic of the program as the way it was written. Therefore, the characterization and performance data presented in later sections are all based on the fully optimized binary with O3 option.

The MPEG-2 decoder used for our case study in Section 5 is taken from Mediabench suite [5], a set of multimedia related applications such as MPEG and JPEG encoding/decoding, audio data format conversions and so on.

4.0 The effect of Algorithm Transformation on General Purpose Processors

This section evaluates several algorithm transformations relevant to the optimizations of program that runs on a general purpose microprocessor. As we mentioned in the introduction, some of transformation is wasteful or even worse since the same functionality is provided by the hardware. We will first analyze how high-level algorithms are mapped or compiled to the real processor by showing the data dependence graph and examine the effect of single assignment, unfolding transformations and SIMD operations.

4.1 Understanding Data Dependence Graph Changes During Program Compilation

Many DSP algorithms consist of a set of manipulation applied on the matrix or multi-dimensional data arrays. The one of the most efficient implementation of those algorithms would be vector or array structure hardware that operates on multiple data elements using single or a set of instructions. Unfortunately, typical general purpose processor does not support those operations. Specifically, the Alpha architecture does not support SIMD operations or multimedia extensions such as MMX or AltiVec [3] that are popular in many current generation microprocessors. Therefore, matrix or array manipulations are written using loop primitives where a set of instructions change loop indices sequentially so that the same operations can be performed on multiple data points repeatedly.

In addition, matrix or array data access requires explicit memory operations. In RISC-type architectures such as the Alpha, all memory communication are performed through only load and store instructions that put/get data from/to the register file and all computations are executed among only register values. The typical latency of those memory operations are 1~3 in many current-generation implementations. This hardware restriction leads to create additional dependences into the original data dependence graph designed from the algorithm.

Figure 3 illustrates these changes in data dependence graph when operations on a matrix are implemented using loop primitives. Unlike the original data dependence graph shown in Figure 3a, the new dependence graph in Figure 3b has additional circles for memory communications since all data communication occur through memory, and also has operations for loop indices update. In this new data dependence graph, independent operations across loop iterations cannot execute in parallel because the loop control-related dependences serialize the whole execution of computations and hence the algorithm cannot achieve its desired parallelism, which leads to performance degrada-

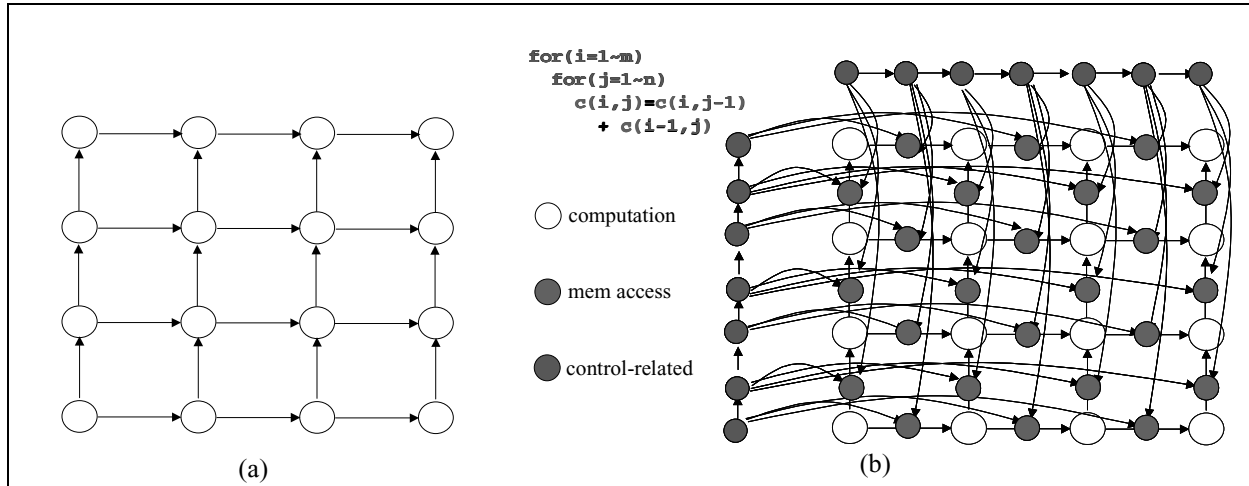
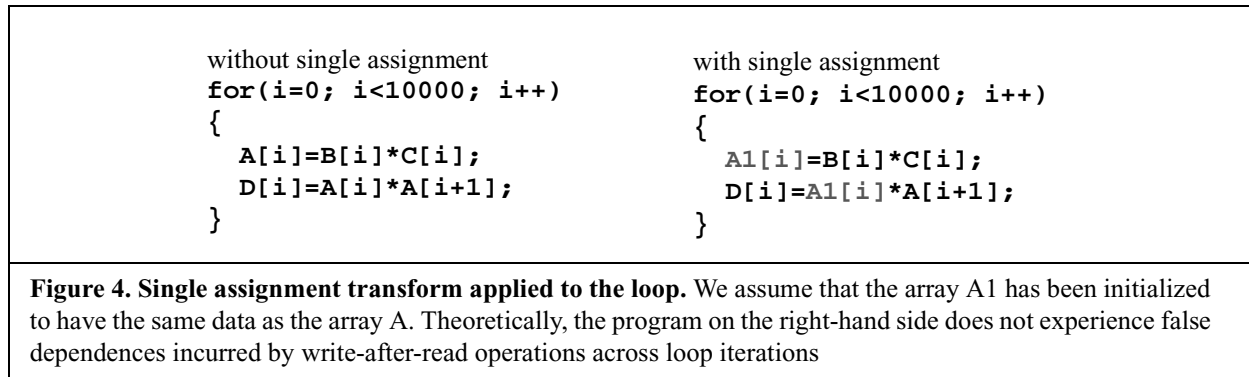


Figure 3. Data dependence graph changes during matrix manipulation mapping to a general purpose processor. (a) represent the data dependences of operations specified in the original algorithm. In (b), memory access and control-related operations are introduced into the graph and they create additional dependences throughout the original dependence graph when the algorithm is mapped using loop primitives.

tions.

4.2 The effect of Single Assignment Transform



Single assignment transform enables the algorithm to achieve its maximum performance by eliminating false dependences incurred by the limitation of storage spaces. Figure 4 shows the example of single assignment transform applied to the loop. The original code on the left-hand side has a false dependence due to write operation to the array A and read operation from the same element across loop iterations. Supposing that the array A1 has been initialized to have the same content of the array A on the right-hand side code that has single assignment transform, it does not experience write-after-read dependence across iterations. The original algorithm does not have these false dependences and all loop iterations can be performed in parallel in both cases. To verify the compiled binaries from these source codes with and without single assignment transformation, we run the both binaries on our infinite machine simulator that keeps track of only real operations without any hardware restriction. We also run the same binaries on the realistic Alpha machine model to measure the performance when they run on the real processor. The

descriptions on our simulator models are presented in Section 3.

Table 1: The effect of single assignment transformation on the execution time

Simulators	With transformation	Without transformation
Infinite simulator measuring theoretical speed limit (algorithm's dependence depth)	2 clock cycles	10001 clock cycles
Realistic processor simulator	152523 clock cycles	141786 clock cycles

The result of performance evaluation is shown in Table 1. As we can expect, the compiled binary with single assignment transformation achieves the theoretical speed bound by performing all operations in each loop iteration in parallel while the original binary with false dependences shows the serialized computations that reflect the number of iterations (10000 times) due to memory location aliasing that makes later write operation wait for the previous read operation to be completed. However, the real performance of two binaries is counter-intuitive. The original code achieves better performance than the transformed binary with single assignment. The reason for this phenomenon can be found from the structure of the Alpha implementation. The alpha 21264 processor dynamically allocates the temporary storage or store queues to memory operations to resolve any false dependences so that it achieves higher performance. The processor does not experience write-after-read dependences regardless of the transformation, which leads to single assignment transformation on the source code or even binary useless. Rather the single assignment negatively affects the performance due to the introduction of additional array A1, since the overall data footprint is significantly increased by this storage and hence it potentially increases the data cache miss rates. The huge performance gap between the theoretical and realistic machines can be explained by hardware constraint such as narrow instruction window, a few functional units, branch misprediction penalties and the effect of cache misses. Specifically, the narrow instruction window serializes dependent operations in the loop since the hardware mechanism cannot detect them given the small scope over the instructions.

Given the hardware restriction, it is more important to reduce the operations due to memory and loop overhead rather than to exploit the parallelism of the algorithm.

4.3 The effect of unfolding

<pre> without unfolding for(i=0; i<10000; i++){ A[i]=B[i]*C[i]; } </pre>	<pre> with unfolding factor 10 for(i=0; i<10000; i+=10){ A[i]=B[i]*C[i]; A[i+1]=B[i+1]*C[i+1]; ... A[i+9]=B[i+9]*C[i+9]; } </pre>
<p>Figure 5. Unfolding transform applied to the loop with independent iterations. The right-hand side code shows that the loop is unfolded by the factor of 10.</p>	

Unfolding transformation is a process of unfolding a loop so that several iterations are unrolled into the same

iteration. When designing a hardware with unfolding transformation, it requires more hardware units but it also enables block processing and reducing sample period by exploiting parallelism across loop iterations. However, we do not have control over the existing hardware since all hardware configuration is fixed. Moreover, the narrow instruction window does not fully allow us to exploit parallelism of loop iterations. Unfolding can be used to reduce the overhead incurred by calculating loop index variables.

Table 2: The effect of unfolding transformation on the execution time (independent iterations)

Simulators	With transformation	Without transformation
Infinite simulator measuring theoretical speed limit (algorithm's dependence depth)	1 clock cycles	1 clock cycles
Realistic processor simulator	37958 clock cycles	67705 clock cycles

We performed the same experiment of the previous section to unfolding. The test code with/without unfolding of independent loop iterations is shown in Figure 5. The performance result is also shown in Table 2. Since all loop iterations are independent, the theoretical execution time is 1 clock cycle and unfolding transformation does not affect this theoretical bound. However, the actual performance on the real machine simulator is significantly improved by the transformation because unfolding reduces the loop overhead and put parallelizable operations near to each other so that the dynamic instruction window can detect those operations.

In order to better understand the effect of loop unfolding transformation, we performed the sensitivity study of unfolding transformation to the machine width as shown in Figure 6. As the machine gets wider, we can see that

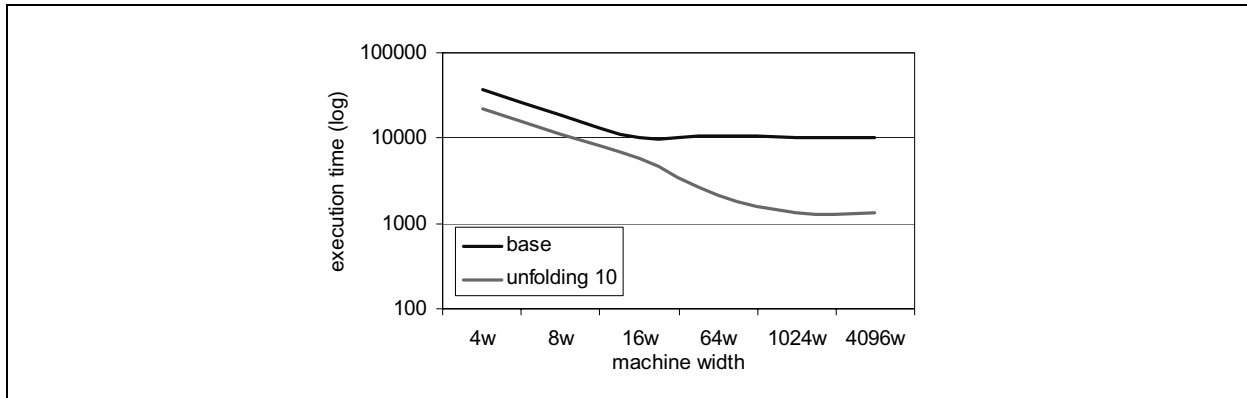


Figure 6. Sensitivity of unfolding transformation on independent iterations to the machine width.

the performance more benefits from unfolding transformation due to less resource restrictions. On 4096-wide machine, we achieve near 10X speedup over the base algorithm whose performance improvement is saturated from 16-wide machine. Although all loop iterations are independent as we discussed, the performance of unfolding by a factor of 10 does not go below 1000 clock cycle since loop iterations are serialized by the loop index variable.

On a narrow machine such as a 4-wide machine, the performance benefit comes from the lower loop overhead while the benefit from algorithm's parallelism becomes evident on wider machines, which is hardly found

among modern general purpose microprocessors.

To better understand the loop control-related overhead eliminated by unfolding transformation, we present the result of the unfolding transformation applied to a loop with dependent iterations.

Table 3: The effect of unfolding transformation on the execution time (dependent iterations)

Simulators	With transformation	Without transformation
Infinite simulator measuring theoretical speed limit (algorithm's dependence depth)	10000 clock cycles	10000 clock cycles
Realistic processor simulator	74335 clock cycles	93650 clock cycles

<pre> without unfolding for(i=0; i<10000; i++){ A[i]=A[i-1]*C[i]; } </pre>	<pre> with unfolding factor 10 for(i=0; i<10000; i+=10){ A[i]=A[i-1]*C[i]; A[i+1]=A[i+0]*C[i+1]; ... A[i+9]=A[i+8]*C[i+9]; } </pre>
<p>Figure 7. Unfolding transform applied to the loop with dependent iterations. The right-hand side code shows that the loop is unfolded by the factor of 10.</p>	

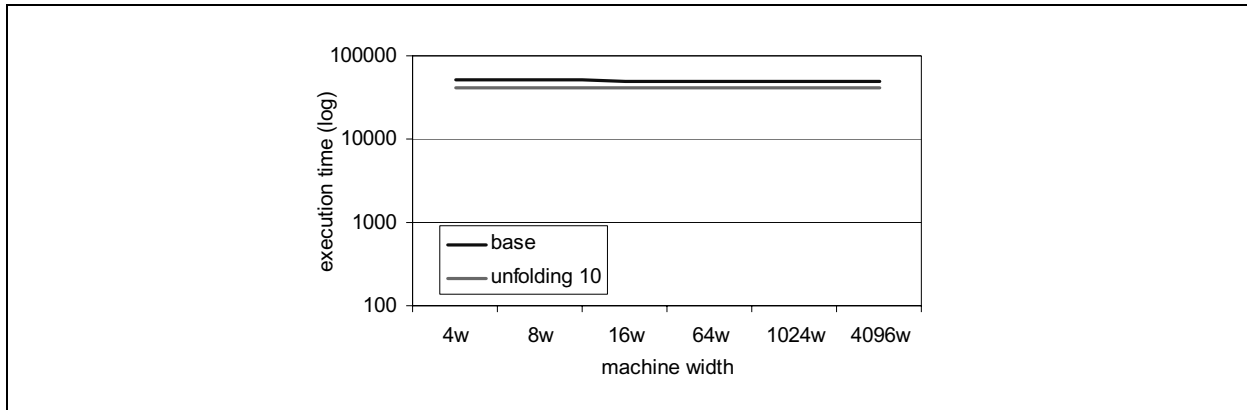


Figure 8. Sensitivity of unfolding transformation on dependent iterations to the machine width.

As shown in Figure 7, all iterations of the loop are dependent on the previous operations and hence the unfolding does not help to achieve parallelism of the algorithm, which is also consistent with Figure 8 where the performance does not change throughout all machine widths. However, since unfolding reduces the overhead incurred by loop index operations, unfolding still shows 20% of execution time reduction comparing against the base code. Note that the Y-axis is drawn in log scale. Table 3 presents the effect of unfolding on an ideal and realistic machine and shows that this benefit is reflected on the execution time of unfolded loops.

However, unfolding does not always help performance due to the limited scope over instructions. If the loop contains relatively more computations than the loop variable overhead, the benefit of unfolding transformation is

restricted only on extracting parallelism by executing multiple iterations simultaneously. Since the Alpha 21264 can detect only 15 independent integer instructions and 20 independent floating-point instructions, the parallelizable instructions outside this window cannot be executed; therefore, the performance improvement is observed in much wider machine and the 4-wide Alpha implementation cannot achieve the benefit from the transformation. The code

Table 4: The effect of unfolding transformation on the execution time (long iteration)

Simulators	With transformation	Without transformation
Infinite simulator measuring theoretical speed limit (algorithm's dependence depth)	4 clock cycles	4 clock cycles
Realistic processor simulator	133983 clock cycles	139642 clock cycles

<pre> without unfolding for(i=0; i<10000; i++){ A[i]=A[i]*E[i]; B[i]=B[i]*A[i]; C[i]=C[i]*B[i]; D[i]=D[i]*C[i]; } </pre>	<pre> with unfolding factor 5 for(i=0; i<10000; i+=5){ A[i]=A[i]*E[i]; B[i]=B[i]*A[i]; C[i]=C[i]*B[i]; D[i]=D[i]*C[i]; } </pre>
<p>Figure 9. Unfolding transform applied to the loop with long iterations. The base code contains 4 times more computations than previous examples</p>	

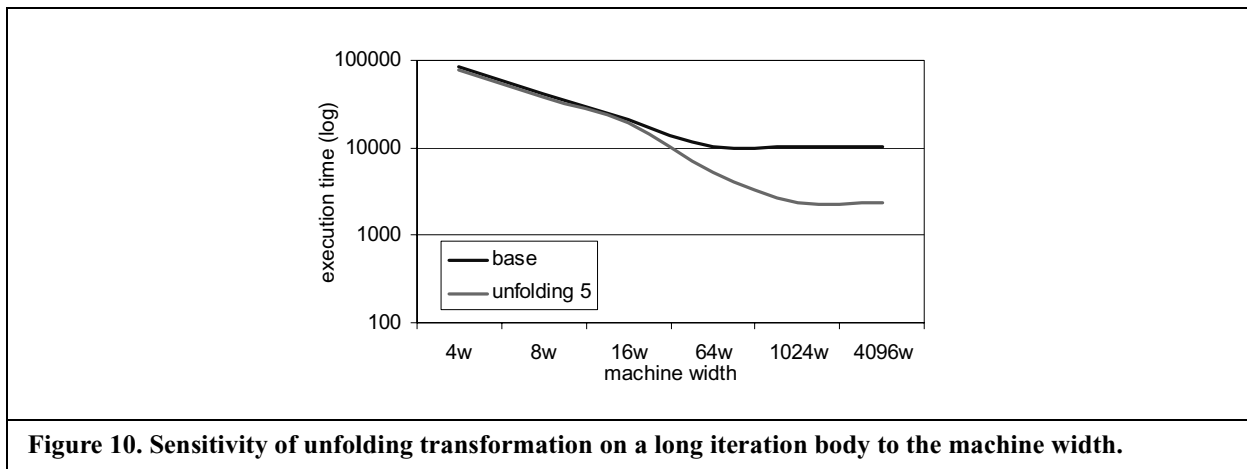
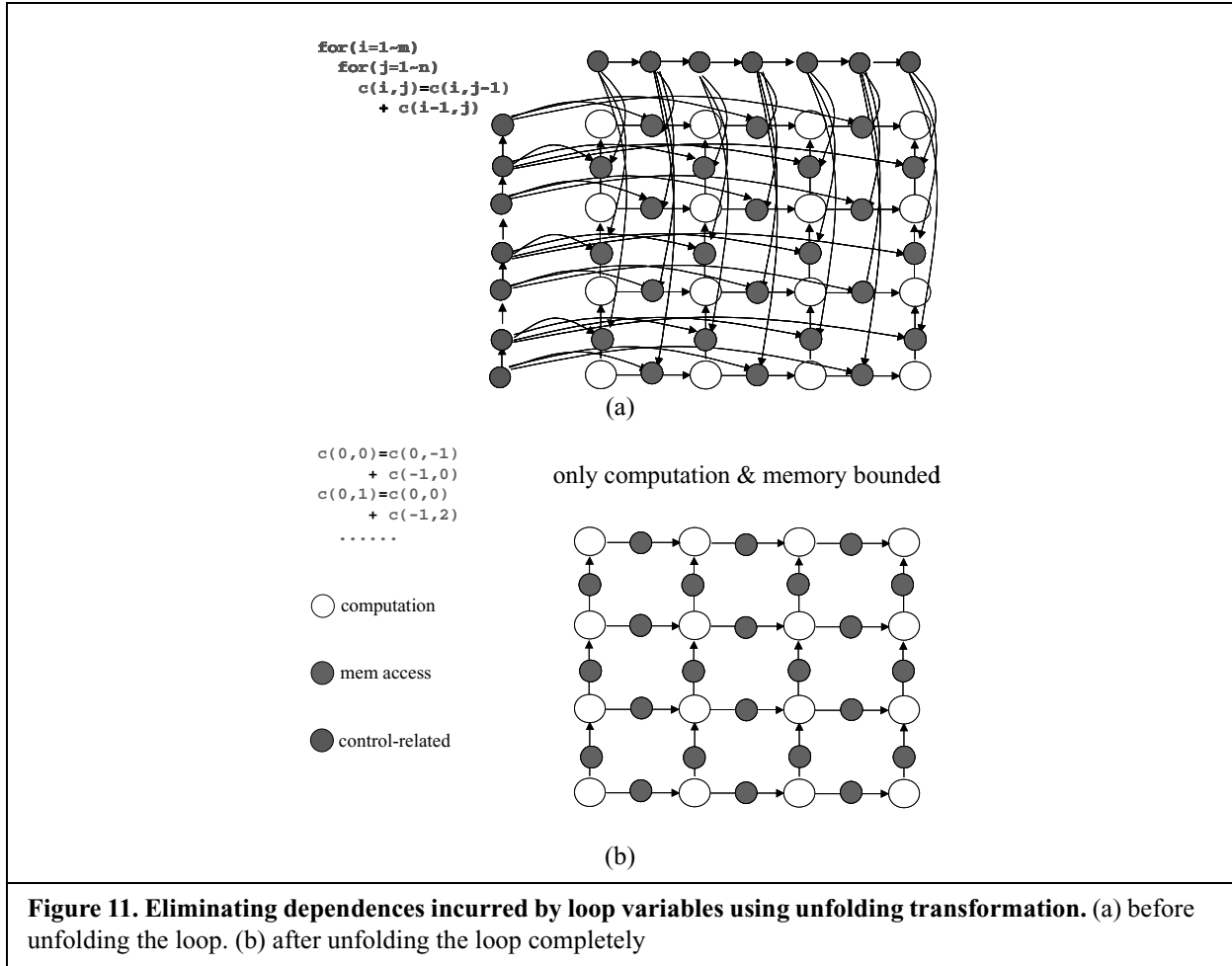


Figure 10. Sensitivity of unfolding transformation on a long iteration body to the machine width.

shown in Figure 9 contains 4 times more operations than previous example in Figure 7. Since the amount of operations in an iteration is greater than the size of instruction window, the performance improvement of unfolding transformation is negligible in narrow machines due to small loop overheads. Figure 10 shows the sensitivity of unfolding transformation on a long iteration body. The performance diverges after 64-wide machine that is able to detect independent operations in the next loop iteration.

In conclusion, unfolding transformation helps to reduce the loop overhead and achieve parallel execution of independent operations across iterations. However, if the iteration contains many computations that do not fit into instruction window, further unfolding transformation does not improve the performance; rather it increases the size of instructions that may end up degrading the performance. Figure 11 shows the data dependence graph analogous to unfolding transformation.



4.4 Reducing memory communication cost by SIMD operations

Although the unfolding transformation is very effective in optimizing loops by eliminating constraint on data dependence and reducing the loop variable overhead, the memory communication dependence imposed by the data communication model of the hardware is not removed. Even though the Alpha architect does not support SIMD instruction extensions, we can apply this to the Alpha binaries by utilizing 64-bit datapath. The basic idea of SIMD operations is shown in Figure 12. In Figure 12a, the original code requires two memory accesses per operations. In Figure 12b, Four short integer accesses are merged into one 64-bit operation. Since the Alpha architecture has more integer functional units than memory unit and integer operations has shorter latencies than memory operations, it improves the overall performance.

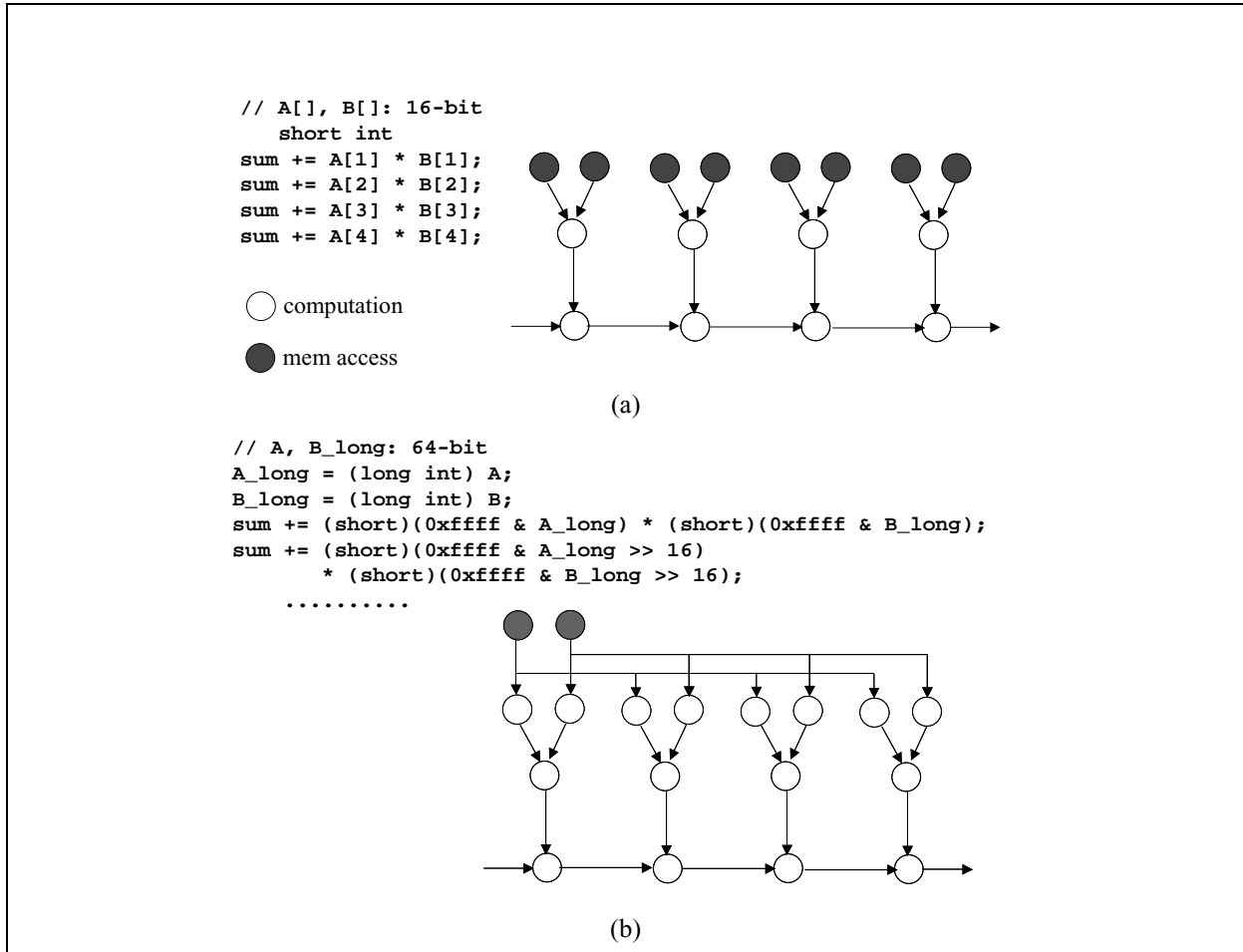


Figure 12. SIMD operations on the Alpha architecture. (a) The original code requires two memory accesses per operations. (b) Four short integer accesses are merged into one 64-bit operation. Since the Alpha architecture has more integer functional units than memory unit and integer operations has shorter latencies than memory operations, it improves the overall performance.

To carefully understand this transformation, we present a sequence of assembly code that represent a loop in Figure 13. SIMD parallelism changes the loop to operate on four 16-bit words at a time. To keep the example simple, we assume that the vector p is quadword-aligned, that the vector length is a multiple of four, and that the length is less than 65536. Our strategy is to load a quadword and split it into two chunks. By splitting the quadword, we introduce 16 guard bits between the data chunks. We then sum each chunk within the loop, accumulating two sums per chunk. On loop exit, we will need to accumulate all of the partial sums.

5.0 Case Study: MPEG-2 Decoder

So far, we have shown the effect of algorithm transformations that can be applied to the loop-intensive DSP applications and possible solutions to reduce loop indices and memory communication overheads considering real hardware constraint of the Alpha architecture. Based on those findings, we show a case study on optimizing a real DSP application, MPEG-2 decoder in this section.

```

(a) source code
unsigned int csum = 0;
{
    unsigned int csum = 0;
    for (i=0; i<length; i++)
        csum +=*p++;
    return csum;
}

(b) compiled binary
loop:
LDWU    $18, ($16)           # load *p
LDA     $16, 2($16)         # p++
LDA     $17, -1($17)        # Decrement length
ADDQ    $0, $18, $0        # csum += *p
BGT     $17, loop

(c) SIMD version
loop:
LDQ     $18, ($16)          # Load *p, 64-bit
ZAPNOT  $18, 0x33, $0       # get the first chunk
ZAP     $18, 0x33, $1       # get the second chunk
SRL     $1, 16, $1          # shift right logical 16 bits
ADDQ    $24, $0, $24        # calculate sum on the first chunk
ADDQ    $25, $1, $25        # calculate sum on the second chunk
LDA     $16, 8($16)         # p++
LDA     $17, -4($17)        # decrement length
BGT     $17, loop
# at the end of the loop, teh partial sums are in $24 and $25, and they need to
be accumulated

```

Figure 13. SIMD parallelism applied on assembly code for a loop

The source code of MPEG-2 decoder is taken from Mediabench Suite [5]. The decoder converts MPEG-2 bit stream into uncompressed video frames based on ISO/IEC DIS 13818-2 codec. Because of the characteristics of MPEG algorithm that performs inverse discrete cosine transform on a small 8 by 8 element blocks, the source code contains many single and nested loops with the small number of iterations (typically 8 iterations per each loop), and each block consists of operations performed on short integer (16-bit) values. We profiled the unmodified decoder using gprof [6], a profile tool that collect the data on how long functions in the program are executed. The profile data shows that the decoder spends 64.8% of the time on *Reference_IDCT* function that performs IDCT multiply operations. Figure 14 shows the source of this function.

This routine consists of two large loop in which each large loop has two nested small loops. The first and the second loop are similar, except that the second loop transposes the index variables so that the routine performs matrix multiplications. We will try two techniques presented in the previous section: first, we try to optimize nested loop by unfolding transformation. Second, we apply SIMD parallelism to avoid unnecessary memory accesses.

5.1 Unfolding transformation

We present the source code changes due to unfolding transformation in Figure 15. In order to minimize the effect of index variable k, we completely removes the inner-most loop and unfolding the 8 iterations into 8 independent operations. Although the operations in the loop is still dependent on i and j variables, further unfolding is unne-

```

void Reference_IDCT(block)
short *block;
{
  int i, j, k, v;
  double partial_product;
  double tmp[64];

  for (i=0; i<8; i++)
    for (j=0; j<8; j++)
      {
        partial_product = 0.0;

        for (k=0; k<8; k++)
          partial_product+=
            c[k][j]*block[8*i+k];

        tmp[8*i+j] = partial_product;
      }
}

/* Transpose operation is
integrated into address
mapping by switching
loop order of i and j */

for (j=0; j<8; j++)
  for (i=0; i<8; i++)
    {
      partial_product = 0.0;

      for (k=0; k<8; k++)
        partial_product+=
          c[k][i]*tmp[8*k+j];

      v = (int)
        floor(partial_product+0.5);
      block[8*i+j] = (v<-256) ? -
        256 : ((v>255) ? 255 : v);
    }
}

```

Figure 14. IDCT multiply routine in MPEG-2 decoder. The profile data shows that the program spends 64.8% of the time in this routine.

```

for (i=0; i<8; i++)
  for (j=0; j<8; j++)
    {
      partial_product = 0.0;

      partial_product+= c[0][j]*block[8*i+0];
      partial_product+= c[1][j]*block[8*i+1];
      partial_product+= c[2][j]*block[8*i+2];
      partial_product+= c[3][j]*block[8*i+3];
      partial_product+= c[4][j]*block[8*i+4];
      partial_product+= c[5][j]*block[8*i+5];
      partial_product+= c[6][j]*block[8*i+6];
      partial_product+= c[7][j]*block[8*i+7];

      tmp[8*i+j] = partial_product;
    }
}

```

eliminating
loop-index
variable k
completely

Figure 15. Unfolding transformation applied to IDCT multiply routine in MPEG-2 decoder.

essary according to our findings in the previous section because the narrow instruction window does not allow parallel executions of long iterations. However, we will present the performance of unfolding that also removes the variable j by unfolding the loop 64 times in the result to evaluate our findings.

The variable *partial_product* may serialize the whole operations because each accumulation is dependent on

the previous `partial_product` variable. However, the Alpha architecture has much longer latency for multiply operations than the add operation, and 8 multiply operations are independent on each other, accumulation operations are not in the critical path of the whole program execution.

5.2 SIMD parallelism

```

long long int longint0, longint1; // prepare for wider references

for (i=0; i<8; i++)
{
    longint0 = tempblock[2*i]; // lower 4 short words (64 bits)
    longint1 = tempblock[2*i+1]; // higher 4 short words (64 bits)
    for (j=0; j<8; j++)
    {
        partial_product = 0.0;
        // extract 16-bit word from pre-accessed 64-bit word
        partial_product+= c[0][j]*((short)(0xffff & longint0));
        partial_product+= c[1][j]*((short)(0xffff & (longint0 >> 16)));
        partial_product+= c[2][j]*((short)(0xffff & (longint0 >> 32)));
        partial_product+= c[3][j]*((short)(0xffff & (longint0 >> 48)));
        .....
    }
}

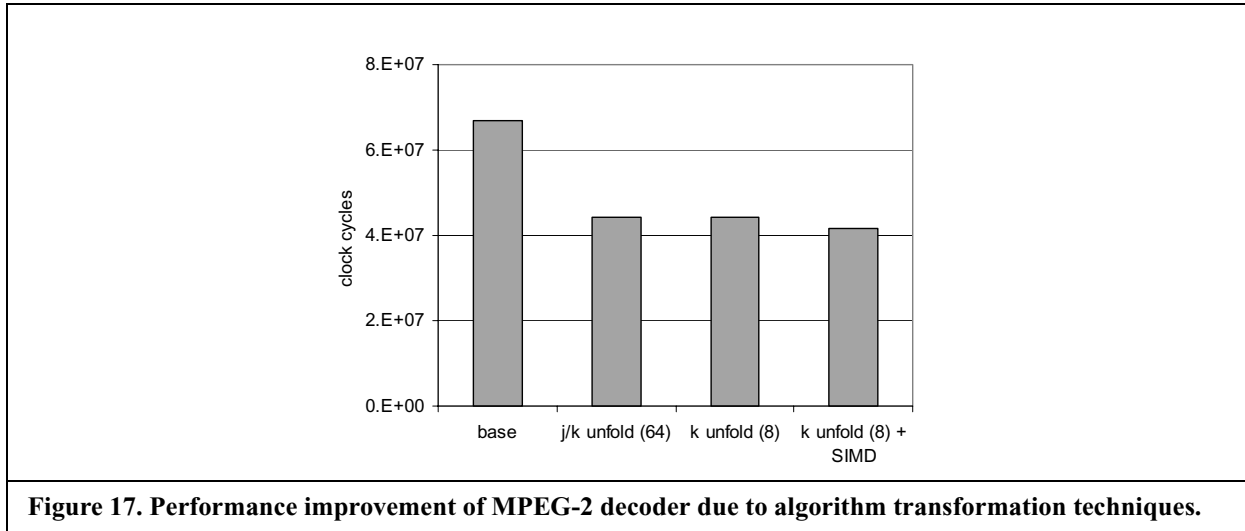
```

Figure 16. SIMD parallelism applied to unfolded loops in MPEG-2 decoder.

To achieve more efficient execution, we perform the SIMD style transformation on this unfolded loop code. Figure 16 illustrates the source code changes that reflect SIMD parallelism. Since *block* array contains a series of 16-bit data taken from the MPEG-2 stream, the memory bandwidth utilization was not efficient in the unmodified function. To utilize 64-bit data path, we introduce two variables, *longint0* and *longint1* that read 64-bit chunks at a time. In the figure, *tempblock* is a 64-bit pointer that facilitates accessing to the original *block* array. Once 64-bit chunks are put into variables, mask and shift operations in the loop split 16-bit values that used to be access by separately. Although mask and shift operations consume more functional units, the bottleneck of the Alpha architecture lies in the memory communications and integer operations have shorter latencies than memory operations, which enables this transformation to get performance benefit from the modified routine.

5.3 Performance improvement over the base MPEG-2 decoder

In addition to *Reference_IDCT* routine, we also modified the source code using transformation techniques presented in the previous sections to maximize the performance benefit. The performance improvement of MPEG-2 decoder due to algorithm transformation techniques are presented in Figure 17. Unfolding that completely eliminates the inner-most loop significantly reduces the execution time by 34.2%. As we expected, the further unfolding does not show measurable performance improvement since the narrow instruction window limits the efficacy of this trans-



formations. SIMD parallelism further reduces the execution time by 5.6% over the 8-unfolded loop, where the combined transformations lead to 38% reduction in the total execution time.

6.0 Conclusions

The importance of efficient mapping of DSP algorithms onto a general purpose processor becomes increasing. Understanding how a general purpose out-of-order processor works helps to improve the implementation of DSP algorithms. Due to serial programming model and hardware limitation, general purpose processors impose memory and loop index variable constraint on the data dependence graph of the original algorithm. To attack these problems, we evaluated various algorithm transformations and presented that unfolding and SIMD parallelism may improve the loop-intensive programs by eliminating the loop index and memory communication dependences. The case study on MPEG-2 decoder shows that these transformations reduces the execution time by up to 38% via efficient algorithm mapping to the Alpha architecture.

7.0 References

- [1] D. C. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.
- [2] Compaq Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.
- [3] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale, *Altivec Extension to PowerPC Accelerates Media Processing*, IEEE Micro, vol. 20, no. 2, pp. 85--95, 2000.
- [4] R. M. Tomasulo, *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal, Vol. 11, pp. 25-33, January 1967.
- [5] <http://www.mpeg.org>
- [6] <http://www.gnu.org>