Spring 2015

Characterization of vectorization strategies for recursive algorithms

Shruthi Balakrishna
Purdue University

Recommended Citation

Balakrishna, Shruthi, "Characterization of vectorization strategies for recursive algorithms" (2015). *Open Access Theses*. 558.
https://docs.lib.purdue.edu/open_access_theses/558

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Shruthi Balakrishna

Entitled
Characterization of Vectorization Strategies for Recursive Applications

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

MILIND KULKARNI

ANAND RAGHUNATHAN

SAMUEL P. MIDKIFF

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): MILIND KULKARNI

Approved by: Michael R. Melloch 04/23/2015

Head of the Department Graduate Program Date

CHARACTERIZATION OF VECTORIZATION STRATEGIES FOR RECURSIVE ALGORITHMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Shruthi Balakrishna

In Partial Fulfillment of the

Requirements for the Degree

of

Master Of Science in Electrical and Computer Engineering

May 2015

Purdue University

West Lafayette, Indiana

To my Dad, Mom, Vishruth and Sukeerth.

ACKNOWLEDGMENTS

First and foremost, I'd like to express my gratitude to my fantastic advisor, Dr.Milind Kulkarni, who has mentored and guided me during my Thesis research. I have had the fortune of starting research with Milind in my first semester here at Purdue. Ever since, I have had a very fulfilling and enlightening experience . He has been a very supportive and inspiring mentor and has definitely played the biggest role in making my research experience here a fulfilling one. His teaching methods, technical discussions during my one-on-one meetings every week and his insights into research have completely changed my perspective about problem solving and research. He has pushed me to think and solve problems while giving me freedom to express my thoughts and ideas in my work. Whenever I am stuck with a problem which I feel is very hard and complicated, I am always amazed at how he manages to provide extremely simple ideas and solutions to overcome them. I'm glad that I got an opportunity to work with him and will be proud of it always.

I would like to thank my professors Mithuna Thottethodi and Dongyan Xu for their courses in architecture and operating systems that gave me skills to conduct my research. I'd like to thank Jack Lee for mentoring me during my Internship at Qualcomm.

Last but not least, I must express gratitude to my family and close friends- here and back in India for their support throughout my life. My parents and brother were always there for me when I wanted to unwind and have been a big part of my support system.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SSE	SIMD Extensions
GPU	Graphics Processing Unit
AVX	Advanced Vector Extensions
CUDA	Compute Unified Device Architecture
BFS	Breath-First Search
DFS	Depth-First Search
DAG	Directed Acyclic Graphs
RAW	Read After Write
WAR	Write After Read
T-Game	3x3 TicTacToe Game
SSR	Single Step Reexpansion
PDF	Probability Distribution Function
CDF	Cumulative Distributive Function

ABSTRACT

Balakrishna, Shruthi. MSECE, Purdue University, May 2015. Characterization of Vectorization Strategies for Recursive Algorithms . Major Professor: Milind Kulkarni.

A successful architectural trend in parallelism is the emphasis on data parallelism with SIMD hardware. Since SIMD extensions on commodity processors tend to require relatively little extra hardware, executing a SIMD instruction is essentially *free* from a power perspective, making vector computation an attractive target for parallelism.SIMD instructions are designed to accelerate the performance of applications such as motion video, real-time physics and graphics. Such applications perform repetitive operations on large arrays of numbers. While the key idea is to parallelize significant portions of data that get operated by several sequential instructions into a single instruction, not every application can be parallelized automatically. *Regular* applications with dense matrices and arrays are easier to vectorize compared to *irregular* applications that involve pointer based data structures like trees and graphs. Programmers are burdened with the arduous task of manually tuning such applications for better performance. One such class of applications are recursive programs. While they are not traditional serial instruction sequences, they follow a serialized pattern in their control flow graph and exhibit dependencies. They can be visualized to be directed trees data structures. Vectorizing recursive applications with SIMD hardware cannot be achieved by using the existing intrinsics directly because of the nature of these algorithms. In this dissertation, we argue that, for an important subset of recursive programs which arise in many domains, there exists general techniques to efficiently vectorize the program to operate on SIMD architecture. Recursive algorithms are very popular in graph problems, tree traversal algorithms, gaming applications et al. While multi-core and GPU implementation of such algorithms have been explored, methods to execute them efficiently on vector units like SIMD and AVX have not been explored. We investigate techniques for *work generation* and *efficient vectorization* to enable vectorization in recursion. We further implement a generic tree model that allows us to guarantee lower bounds on its utilization efficiency.

1. INTRODUCTION

1.1 Motivation

The transistor limit reached for designing serialized, traditional central processing units(CPUs) resulted in devoting increased transistor counts to achieve parallelism with multiple cores. Successive generation of processors evolved by increasing their core counts and scope of parallelism. While shared-memory parallel programming is not trivial, many programming models and execution platforms have been developed that allow programmers to write efficient programs for these machines. Unfortunately, due to concerns about power consumption and energy efficiency, many vendors have started developing throughput-oriented devices. Rather than providing a handful of general-purpose cores, throughput oriented devices provide vastly more, but simpler, computational cores. These can take the form of vector units attached to general purpose cores such as the streaming SIMD extensions (SSE) or advanced vector extensions (AVX) or separate accelerators that are effectively large vector machines, such as GPUs. The common thread for all of these is that they provide substantially more parallelism than general-purpose multicores. Intel's newest Haswell chips have 4 cores per chip, while the AVX2 SIMD units attached to each core provide 8-wide vector parallelism, and NVIDIA‘s latest Kepler GPUs have over 2500 CUDA cores.

The price to be paid for such vast parallel computing power is that programming for these throughput-oriented cores is not as general and direct as general-purpose processors. The programming models tend to be restrictive and vary in terms of the programming language and approach. Their throughput efficiency is a result of certain restrictions imposed on the kind of operations, data distribution and memory dependencies. For example, SIMD (Single Instruction, Multiple Data) vector units are typically programmed using intrinsics, while SIMT (Single Instruction, Multiple Thread) GPUs use low-level, data-parallel languages such as CUDA [1].

Moreover, extracting sufficient performance out of vector-based cores requires carefully structuring code and data. While each core on a general-purpose multicore can run an independent thread, to fully exploit the 8-way parallelism offered by AVX2, a programmer must find 8 identical operations that operate on different data, and then must be able to load that data efficiently into SIMD registers, which requires ensuring that the data is contiguous in memory. Though the GPU performance model is somewhat less fragile, efficiently exploiting SIMT parallelism requires identifying threads that are performing much of the same work, and operating over nearby data.

The result of these programmability issues is that most research in programming vector units and GPUs have taken one of two tacks. One option is domain-specific programming models, such as ones focusing on dense linear algebra, where data organization is straightforward and computations are regular and predictable and hence can be vectorized easily. Another option is to invest substantial programmer time to code, debug and tune hand-written implementations. In addition to the lack of generality, both of these options suffer from another critical drawback: they do not offer portability. They require learning new programming models to target throughput architectures; there is no portability to other architectures, such as general-purpose multicores.

The key challenge in effectively vectorizing programs is to account for the two important restrictions of vectorized execution models. First, to vectorize multiple operations, the instructions must be the same; any system for vectorization must therefore be able to efficiently find identical computations to execute in parallel. Second, a primary cost in performing vector operations is marshaling data that the operations require (e.g., loading the operands into a register); an effective system for vectorization must therefore organize data so that it can be easily manipulated by vector operations.

We consider a vector machine to be a set of P processors. These P processors can each execute a different thread, with the following restrictions. First, the processors only execute in parallel if they are performing the same operation. If only a subset of the processors are performing the same operation, the other processors remain idle until the operation is complete. We call this situation underutilization. Second, if multiple processes are performing a load, they are not guaranteed to execute in parallel, though the operations are the same. Loads only proceed in parallel if they are either loading the same location, or are loading from "nearby" memory locations. Hence, only if all the data that has to be loaded is in the cache, they can operate in parallel.

To see why exploiting parallelism on such a machine is difficult, consider the following naive strawman approach to running a program with P threads. As in a multicore system, each processor runs a single thread and each thread runs independently, tracking its own program counter, stack, etc. It is clear that there will not be much parallelism available, as there is no reason that threads would be performing the same operation. Moreover, even if threads are performing the same operation, simply managing the thread stacks reduces parallelism. If every thread is at a different point in its stack, then there is no way to lay out the stacks (e.g., by striping them) so that stack accesses (e.g., to load local variables) are "nearby" and can be done in parallel. Hence, in order to run multiple tasks on a vector machine in parallel, more care must be taken to organize the tasks' execution. This is straightforward in the case of simple parallel for loops: each loop iteration becomes a task, the tasks are all identical, and there is no issue with stack management as the loop body does not have method calls. Unsurprisingly, parallel for loops are the source of most vectorized computation. But parallel for loops are a limited programming model.

To expand the scope here and allow programmers to write more general programs that are portable, vectorizable and efficient, a programming abstraction model is necessary. However, such an abstraction model must address various kind of applications efficiently. In my research, I explore efficient ways to vectorize one such class of applications- *recursive algorithms.* Recrusive algorithms are definitely more complex than parallel for loops and exhibit inherent data and control dependencies. Hence, paralyzing recursive algorithms is a challenge in itself. Vectorizing such functions require analysis of its dependencies and a novel method to vectorize them. Hence, in my research, I work towards developing a simple, abstract vector machine model that captures the limitations of existing vector architectures such as SIMD units, but abstracts away the specifics of each of these architectures and their different low-level programming models for recursive functions.

Recursive algorithms are very popular in many tree traversal problems, gaming applications et al. Many n-player games employ recursive algorithms by following zero-sum, non-zero payoff strategies. A recursive game is defined as a "finite set of game elements, which are games for which the outcome of a single game (payoff) is either a real number, or another game of the set, but not both" [2].Games like chess, tic-tac-toe and n-queen are implemented with recursion. While multi-core and GPU implementation of such algorithms have been explored [3], methods to execute them efficiently on vector units like SIMD and AVX has not been explored. Hence, the parallization model and implementation of vectorization for such recursive functions is explained in this thesis.

1.2 Vectorization of Recursive Tree Traversals

1.2.1 Modelling Recursion as a Computation Tree

Vectorization of recursive algorithms is achieved by first modeling the recursive algorithm as a tree building or tree traversal problem. Every parent-child recursive call can be imagined to be a parent-child node pairs of a tree. The edges indicate that the base case fails for the parent and hence a child node is spawned. When the base case is true, the current node terminates as a leaf node. With this analogy, every recursive algorithm can be imagined to be a tree traversal problem. The number of recursive calls invoked by a parent indicates the number of children per node. The edge weights and node contents are modelled based on the base case and induction cases of the recursion function.

1.2.2 Parallelism in Recursion

Several techniques have been explored to parallalize recursive algorithms [4] [5],, [6]. A popular linguistic and runtime technology for multi-threaded programming is Cilk [7]. In CILK, the work stealing algorithm spawns new threads to parallelize multiple recursive calls within the parent recursion function. All the spawned threads must return before the parent function returns. Hence, for a system with P cores, upto P threads can be executing at any time. CILK and most other existing techniques explore ways to exploit parallelism from recursive functions in one of these ways-

- By operating on disjoint sections of data [5]
- Fine-grain parallelization techniques for recursive calls on multi-core and multi-threaded architectures [4] [8]
- CUDA based automatic parallelization [6].

While the above techniques exploit task level parallelism in recursion or use GPU units, not much research has been done on parallelizing trees for SIMD architectures. The techniques that I propose in my research explores data level parallelism using vectors for SIMD execution. Since recursion does not expose data parallelism directly, we need a scheduler that transforms the existing recursive calls to vectorizable problems that can utilize vector units.

1.3 Vectorization of MinMax trees

Minmax algorithms are decision making algorithms used in many applications of game-theory and statistics. It's main idea is to minimize one player's loss while maximizing other player's gain, given a set of game conditions. The Minmax theorem was worded by John von Neumann states that “For every two-person, zero-sum game with finite strategies, there exists a value V and a mixed strategy for each player, such that

1. Given player 2’s strategy, the best payoff possible for player 1 is V

2. Given player 1’s strategy,the best payoff possible for player 2 is -V. ” [9].

As explained in the theorem, one player (say 1)’s strategy guarantees him a payoff of V regardless of the other Player(say 2)’s strategy, and similarly Player 2 can guarantee himself a payoff of -V. Hence, in this zero-sum game approach, each player minimizes the maximum payoff possible for the other while he maximizes his own minimum payoff. MinMax was initially used in games covering cases where players take alternate or simultaneous moves and later extended to more complex games and decision-making problems. It employs recursion for evaluation and provides a good starting point to study vectorization of recursion. I say so because games offer wider scope of tree structures and can involve extensive computation for predicting strategic moves. Hence, in my research, I begin exploring Recursion Vectorization by implementing the vectorization model for a min-max based 3x3 tic-tac-toe game. This provides a good starting point to experiment vectorization and verify its effectiveness.

1.4 Vectorization of Generic Binary Trees

The scheduling techniques applied to minmax algorithms prove the efficiency of vectorization for a very limited set of recursive trees. Minmax trees are usually bushy trees and exhibit a consistent pattern in its tree shape. It is very rare to find very stringy trees, or trees with both bushy and stringy sub-trees together in such games. Hence, to prove the credibility of these techniques, it is important to extend the experimentation to other more generic trees. In my research, I do this by building a variety of random binary trees to profile for the effectiveness of vectorization. These trees can represent non-SIMD compatible programs and provide a good platform to test the scheduling policies. The shape of these DAGs are controlled with two parameters - number of nodes in a tree and the maximum depth of the tree. Varying these parameters result in diverse non-SIMD trees which can be vectorized using the scheduling policies proposed in the future chapters. We hope that this exercise of designing generic trees to test the scheduling policies will provide us with insight that would enable us to design universal strategies that provide good bounds for a large subset of computations.

1.5 Contributions and Organization

This thesis explores the challenges involved in vectorizing recursive function and proposes scheduling techniques to enable vector operations for such functions. The primary contributions are:

1. Development of necessary conditions for vectorization and analysis of recursive algorithms to determine the scope of its vectorization.

2. Implementation of a *Work Generation* transformation to create a starting work-vector that can utilize vector units for recursive functions.

3. Development of *restart* and *re-expansion* scheduling policies to improve utilization of vector operations.

4. Build and analysis of generic binary trees using the new scheduling transformations to expose SIMD opportunities for a larger subset of recursive problems.

2. BASELINE SCHEDULING MECHANISMS FOR VECTORIZATION

In this chapter, we describe the basic transformations and scheduling mechanisms to extract vectorizable parallelism from SIMD-incompatible programs. In particular, these transformations will be tested by transforming a two-player 3x3 Tic-Tac-Toe game(will use the term T-game interchangeably for brevity) into a vectorizable form.

2.1 Challenges that limit Vectorization

Transforming recursive functions to generate vectors that can be executed in vector machines is based on the following two ideas.

1. *Pseudo-Tail Recursion*: In short, for vectorization, bottom-up traversal of any part of the recursion tree must not be allowed. Recursive functions, due to their nature may require a recombination step at its parent node after computation. Hence, bottom-up traversal of the tree may be required. This creates write-after-read (WAR) dependencies in the tree. The vector nodes are forced to access individual parent nodes which are distributed at different locations in the memory layout. This introduces additional overhead and inefficiency in the scheduler. Moreover, this operation is again non-SIMD compatible. Creating a scheduling technique to generate SIMD-Compatible code using non-SIMD compatible instructions as part of the new code clearly defeats the purpose. Hence, recursive functions must be transformed to a pseudo-tail recursive form before vectorizing them.

2. *Work Generation*: Sufficient work (work and nodes are used interchangeably) should be available to form a vector-block. Each node in this block must have

the same stack depth and execute the same computation. Recursive trees traverse in a depth-first fashion naturally. Hence, at any point during its execution, only one node is in its computation state. In some cases, independent sub-trees can execute their respective nodes in a parallel thread. But determining such nodes for every vector-formation is not feasible. Also, it requires one pass of the tree prior to its actual traversal to determine dependencies. Hence, there must be a generic work-generation function that creates a vector-block for SIMD processing.

2.2 Pseudo-Tail recursion

2.2.1 Modifying recursion into pseudo-tail recursion

To achieve data parallelism, the recursion function is modified such that there are no WAR dependencies. This is done by converting the existing recursive function to a pseudo-tail recursive function [10]. A pseudo-tail-recursive function is a recursive function where there are no operations that follow the recursive call. The recursion function call in the last step of the function. In other works, a recursive function call's successor in a control-flow graph is one of these two types of block -it is either a base-case-true exit node's function call, or is another recursive function call. All operations that occur after a recursive function call are either pushed to the children nodes or to the leaf nodes. In other words, these computations happen before the next recursive call is invoked or after all the recursive calls are completed and the base case of recursion is encountered.

In the code shown in Figure 2.1, the base case and inductive case are clearly distinguished with the if-else condition. The base case returns a final value. This value is used to evaluate results based on the application in *after_recurse_evaluation* function. Clearly, the *after_recurse_evaluation* function is a post-evaluation function that is dependent on the result of the recursive call. To modify this into a pseudo-tail algorithm, this post-evaluation function must be eliminated. The recursive call must

```
int recurse(node n)
  if (isBase(n))
    return baseCase();  //Returned result is the leaf value
  else
    for all( child c : n)
      leaf_val = recurse(c)
      //Computations after the recursive call returns
      after_recurse_evaluation(result,leaf_val)
    return result
```

Fig. 2.1. A Simple Recursive Function Structure: The function call *after_recurse_evaluation* invoked after the *recurse* function prevents efficient vectorization

Figure 2.2, the method *after_recurse_evaluation* seen in Figure 2.1 is replaced with *update_evaluation* function in the base case. Imagine this to be equivalent to pushing the function from the parent node to its children. *update_evaluation* tracks the recursive calls and transition values and uses a global data structure to store intermediate values sometimes. These store values are used to evaluate the final results of recursion using innovative techniques.

be the last instruction in the control flow. This is done by re-modelling the function call to the form shown in Figure 2.2.

2.2.2 Transformation of T-Game into a Pseudo-tail minmax function

We start by solving the problem of vectorizing recursive algorithms for an example application-The TicTacToe game(abbreviated as T-Game) . A computerized T-Game generates all possible moves for a given board state and evaluates the chances of a player's victory for each possible move. A popular implementation of this algorithm is using the min-max approach. In order to vectorize this game, parallel tasks must be identified. For recursive tree building, one must first modify the algorithm to ensure that all the RAW dependencies are eliminated. This is a requirement to ensure that

```
int recurse(node parent, node n)
  // Computations are pushed to the children
  update_evaluation(parent, n)
  //Use data structures to store any data required for pseudo-tail
  store_temp_data(parent, n)
  if (isBase(n))
    // Use the temp_data to calculate new leaf value
    result = evaluate_result_with_temp_data(n)
    return baseCase(result)
  else
    for all( child c : n)
      recurse(n, c)
```

Fig. 2.2. Pseudo-Recursive Function: The original recursive function modified to strip the tail function in the inductive piece of the code.

point-blocking techniques can be applied to tree traversal efficiently for vectorization. To do this, let's understand the original recursion in brief.

How does TicTacToe use minimax?

During a given turn, assuming player 1 is making a move. A search tree is generated with the code in Figure 2.4, starting from the current position up to the game-end position. Next, from player 1's perspective, since his strategy is to maximize the payoff, he has to pick the move with the maximum payoff among its children. To determine this, inner node values of the tree are filled up in a bottom-up fashion using the leaf node's values. The nodes that belong to player 1 receive the maximum value of its children's nodes. Nodes for player 2 will receive the minimum of its children's nodes. This choice of moves that happens in a bottom-up manner is illustrated in Figure 2.3.

At the leaf level, node values are stored assuming player 1 ends the game. One level above, since player 2 must have made a move, we assume that player 2 would pick the

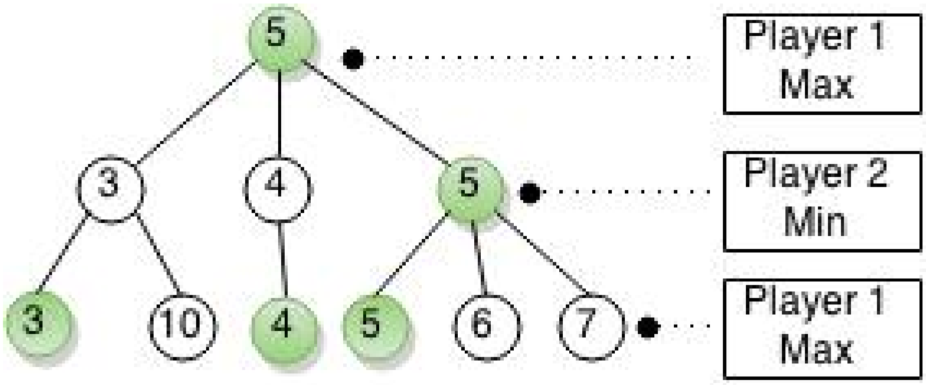


Fig. 2.3. MinMax illustrated

move that works best: i.e. the minimum payoff child node. Next, player 1 will pick the maximum-payoff child node, one level up. This illustrates the minimax behaviour using recursion. The pseudo-code for this implementation is shown in Figure 2.4. Here, the function *minmax_evaluation* performs the minimax comparisons.

Transformation Details

Since minmax uses a bottom-up approach for updating inner node weights and determining the best next move, it modifies the values of the inner nodes after the complete traversal of the tree. Hence, vectorization of such a tree is not possible using a direct technique. Such a tree should be transformed into a pseudo-tail recursive tree. This is achieved by rewriting the eval function to determine the payoff value for each move at the leaf level and keeping a score board to record these values in *update_stats*. The eval function is specific to the application and labels values to the leaf nodes based on the board state and criticality of win or loss. If the current player and the eval function's winner are the same, then a positive weight is stored in the scoreboard. If they are different, negative values are stored. Finally, the main function uses this scoreboard in *pick_best_move* to decide the next player's best move. This is analogous to the min-max algorithm and enables the recursive function to be re-written as a psesudo-tail function.

```
void game_start(board a, root n)
  int best_move_so_far=none;
  while(!game_ended(a))  //Till board is full or won
      for all(move m : possible_next_move(a))
        res = minmax(a_copy, m, status);
        //Try all moves and pick the best move
        if (minmax_evaluation(res,best_move_so_far))
          best_move_so_far = res;
  make_move(a,best_move_so_far);
  return;

int minmax(board a, node next_move, gamestatus status )
  if(game_ended(a))
    return minmax_val(status); //Leaf-node determines win/loss/draw
  else
    make_move(a,next_move);
    for all(move m : possible_next_move(a))
      val = minmax(a,m,status) //Recursion
      minmax_evaluation(val,status); //Post-function evaluation
    return status.bestmove;
```

Fig. 2.4. T-Game with Minmax logic: The simplest form of this algorithm involves invoking a recursive call for every possible move of the board, at each level of the game tree. This is not pseudo-tail recursive.

2.3 Work Generation

2.3.1 Breadth-first execution to extract data parallelism

Now that we have a recursive tree that can be vectorized, we need a vector to start with. So the next question to address is that given a recursive tree, how do we vectorize the tree to efficiently use vector units like SIMD? To achieve this, we define *Work Generation* as the next step for vectorization of recursive algorithms.

```
 void game_start(board a, root n)
  while(!game_ended(a))
      for all(move m : possible_next_move(a))
        res = minmax(a_copy, m, status);
      make_move(a, pick_best_move(game_stats));
    return;
int minmax(board a, node next_move, gamestatus status )
  if(game_ended(a))
    update_stats(game_stats,status) // All computations are done here
    return;
  else
    make_move(a,next_move);
    for all(move m:possible_next_move(a)
      return minmax(a,m,status) //Pseudo-tail property satisfied
```

Fig. 2.5. T-Game with Pseudo-Minmax

After transforming the application into a pseudo-tail recursive application, we extend the idea of Breath-First Tree Traversal to generate sufficient work.

Our proposed scheduling approach for creating work is to perform a breadth-first expansion of the recursive tree shown in Figure 2.6. That is, starting from the initial *recurse* call, we will execute the program in a breadth-first manner. When there is sufficient work, we design our scheduler to carefully control the execution so that the computation tree is executed in a level-by-level manner from then on.

As shown in Figure 2.7, in order to generate multiple nodes for vectorization, a breath-first traversal is performed starting from the root node till an intermediate depth say BFS-D. At BFS-D, all the nodes available are used for vectorization and the function *dfs_recurse* is invoked. These nodes are grouped to form a vector block and this vector block can be processed to simultaneously work on all the nodes using SIMD hardware. To effectively perform SIMD processing, the following steps are followed for point-blocked code generation.

```
int recurse(node parent, node n)
  // Computations are pushed to the children
  update_evaluation(parent, n)
    store_temp_data(parent, n)
  if (isBase(n))
    // Use the temp_data to calculate new leaf value
    result = evaluate_result_with_temp_data(n)
    return baseCase(result)
  else
    for all( child c : n)
      recurse(n, c)
```

Fig. 2.6. Pseudo-Recursive Function

```
int bfs_recurse(Block b, node parent)
  Block next;
  if(height <= BFS_D)
    //Compute children nodes for all nodes in the block 'b'
    for each ( Node n : b)
      update_evaluation(parent, n)
      store_temp_data(parent, n)
      if (isBase(n))
        // Use the temp_data to calculate new leaf value
        result = evaluate_result_with_temp_data(n)
        return baseCase(result)
      else
        for all(child c:n)
          next.add(new c);
    bfs_recurse(n, c)
  else
    dfs_recurse(n,c)
```

Fig. 2.7. Work generation for the Pseudo-Recursive Function

Let the number of nodes present after BFS traversal till depth D be N. For a SIMD width of W, the first W nodes out of the N nodes are grouped together. To perform vectorization, we perform a depth-first traversal of the whole vector instead of individual nodes till all the nodes are processed. After a vector-stack has completed processing, the next set of W nodes are picked up from the pool of N nodes. This process continues till all the N nodes are processed in N/W vectors stacks. Theoretically, the computation time at each depth of the tree gets reduced by a factor of W, excluding the overhead of vectorization and rearranging of the vector nodes. We'll discuss more about the overhead costs in a later chapter.

2.3.2 Evaluation Parameters

To profile the behaviour of vectorization in recursion, we define a few parameters. These parameters collectively provide an estimate of the efficiency and overload costs involved in the technique. Each of these parameters is defined below.

Utilization

We define SIMD utilization of a program run on a SIMD vector as the fraction of SIMD lanes that are kept busy with active data during the run of a program. Average Utilization is the arithmetic mean of every SIMD operations‘s utilization. This is the foremost important parameter in the analysis of vectorization since it represents the percentage of vectorization achieved as a result of the scheduling operations. Ideally, we aim to achieve 100% for each vectorization operation. However, trade-offs are made based on the overhead involved in generating sufficient and continuous work, gains involved in scheduling vectors as opposed to serial execution and the overall gain in a program's efficiency against its serial model.

Under-Utilization

Under-Utilization is the measure of inefficiency in the vectorization process and is measured by tracking the fraction of SIMD lanes in the vector that are inactive during the run of a program. Not every SIMD operation is always full. SIMD vectors can get executed with a few empty lanes. However, the aim of my scheduling policy is to keep this as low as possible or below a certain threshold.

Fullness

Fullness is a measure of the shape and structure of the tree. Imagine an ideal tree to be a fully balanced tree, where every parent node at a given level has fixed number of children. Such a tree has no nodes missing and no space for adding new nodes to the tree. Such a tree is 100% full. Now, to keep track of how many nodes are *missing* from the tree, we define the fullness parameter as the ratio of total nodes in the current tree to the total nodes possible for the given tree, in compliance with the given algorithm. In essence, we want to know how close we are to having a tree where each point does exactly the same thing. This is similar to utilization, but with a subtle focus on understanding how various shaped trees behave when vectorized. For example, a low percentage of fullness indicate *stringy* trees, while higher values refer to *bushy* trees. It is also helpful to keep track of the height of the tree. The ratio of the height of the tree to the number of nodes traversed is another measure of fullness.

2.3.3 Work Generation for Minmax

For minmax application, after rewriting it into a pseudo-tail recursive algorithm, the tree can be vectorized using the approach explained above. In my research, I started with a 3x3 board and followed all the rules of tic-tac-toe. The recursive tree for tic-tac-toe has the structure shown in Figure 2.8.

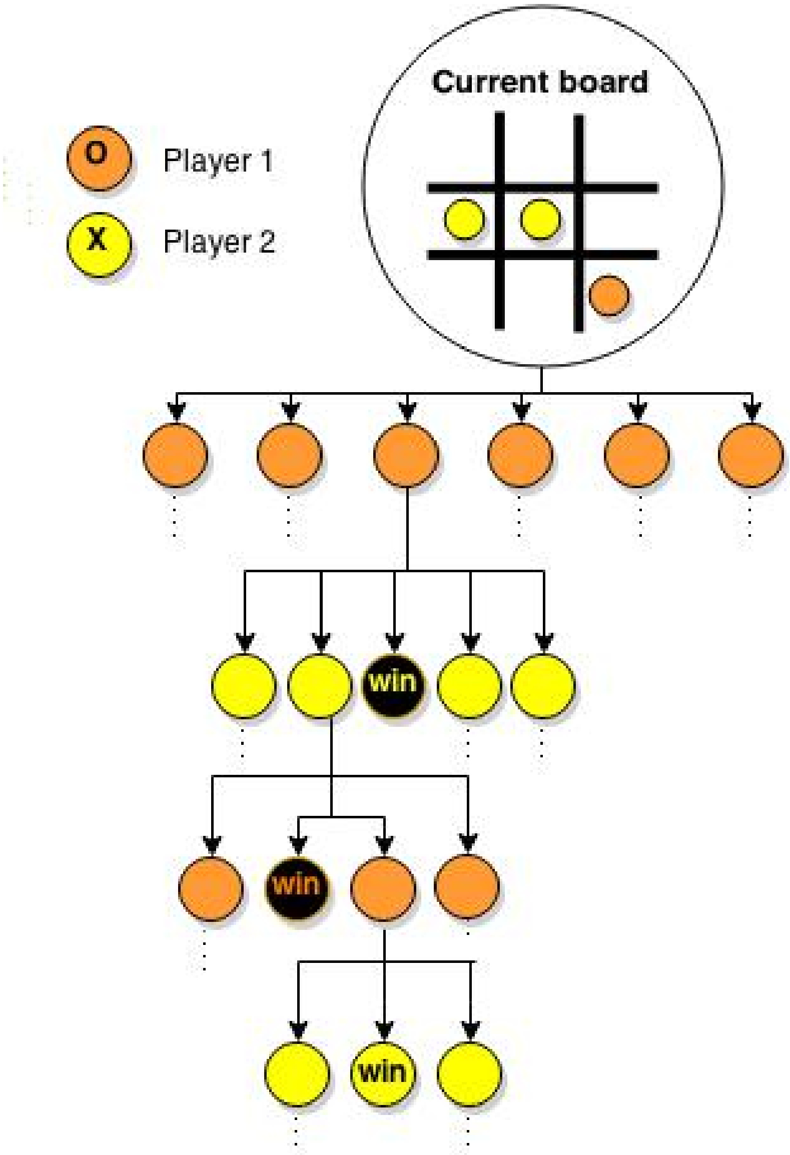


Fig. 2.8. TicTacToe Recursion Tree Structure: A sample tree for a 3x3 board is shown above. The number of children for any given parent decreases with the increase in the tree height. If any instance of the board terminates in a win or draw, the node corresponding to it becomes a leaf node.

If BFS traversal is executed till a depth of say 3, the resulting nose buffer will have a width of 8*7*6 = 336 nodes. If the vector unit's width was greater than 336,

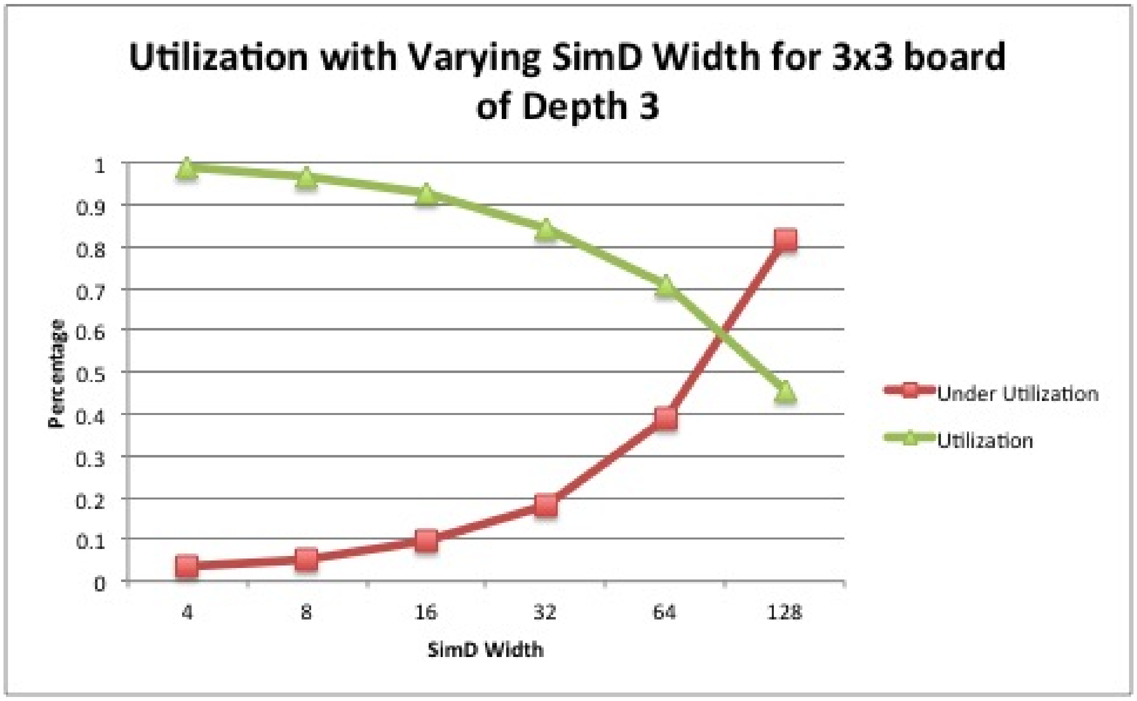


Fig. 2.9. SIMD Performance: Utilization and Underutilization ratios for various SIMD widths using the abstract vectorization model for Minmax T-Game. BFS-D is set to a depth of 3.

all these nodes can get executed in parallel in a depth-first fashion from here on, till the end of the tree. If not, these nodes get executed in blocks , with each block equal to the size of the SIMD vector. The total number of nodes reduces further down the tree as many board-states terminate according to game rules. Hence, as the nodes reduce, we ensure that the buffer gets compacted again by regrouping the scattered nodes into a continuous stream before the next vector operation. This is called stream compaction and is a popular technique used in SIMD and GPU vector operations [11]. This improves the utilization ratio of the vector units and reduces its under-utilization.

To see the effect of vector width and BFS depth on this application, I ran a few experiments. It is important to understand that the performance of vectorization will depend on multiple factors. Some of them are the tree size, SIMD Width, BFS-D and tree shape. For a T-Game application, tree size and shape is predictable

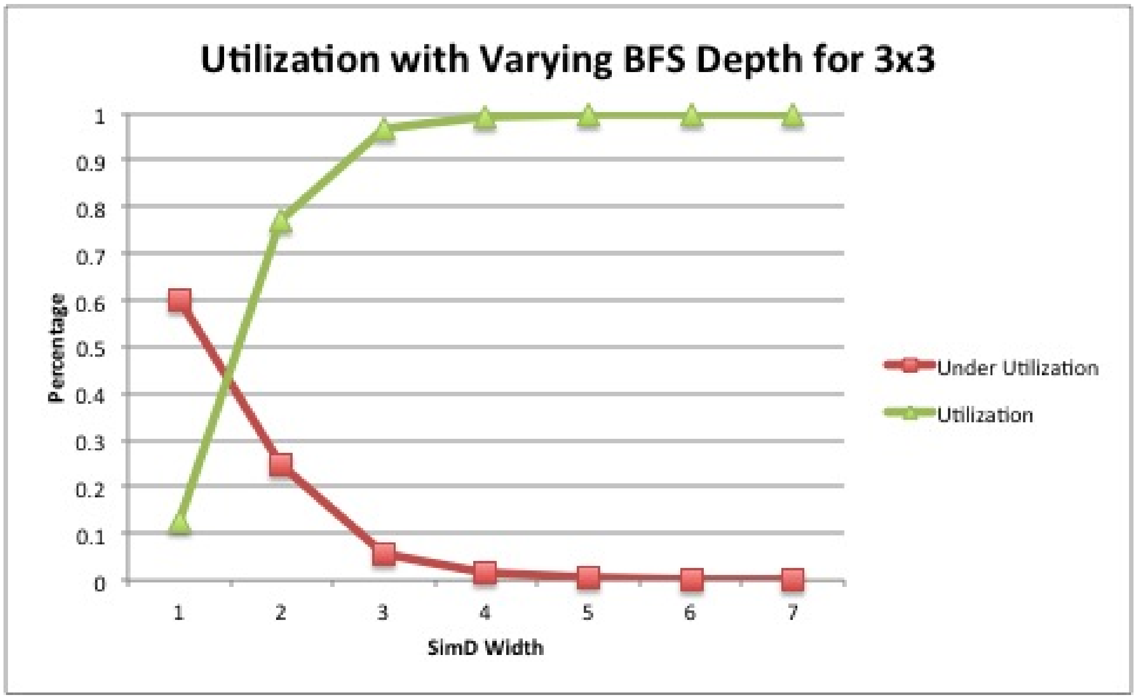


Fig. 2.10. SIMD Performance: Utilization and Underutilization ratios for varying BFS-D values using the abstract vectorization model for Minmax T-Game. SIMD width is set to 16.

and somewhat fixed. Hence, by manipulating the vector width and work generation depth, the performance of vectorization is profiled. The graphs below illustrate the dependence of vector utilization and under-utilization for various vector widths and BFS depths.

As seen in the Figure 2.9, for a 3x3 board with BFS-D depth of 3, there are sufficient number of points to exploit SIMD parallelism for lower SIMD widths. As SIMD width increases, since there are insufficient points, its utilization drops.

Hence, increasing the BFS depth should result in better SIMD utilization since more nodes are available to work with at any point of time. However, it is to be noted that increasing BFS results in more time and space overhead to generate larger blocks. Depending on the SIMD width, a suitable BFS-D is heuristically chosen for a given recursion tree. As shown in Figure 2.10 In case of a 3x3 board, utilization saturates after a depth of 4 and hence does not reflect the impact of increasing BFS-

D. These numbers show significant improvement when higher dimension boards are used with a larger BFS value for the T-Game.

2.4 Conclusion

Work Generation enabled vectorization of recursive tree traversals and provided a means to use SIMD hardware for recursive serial functions. However, just generating an initial vector is not sufficient to sustain a good average utilization ratio. Efficient vectorization requires continuous monitoring of the vector data and ensuring that the vectors get refilled with more work as its size dwindles and reduces utilization. The next chapter deals with mechanisms implemented to sustain efficient Vectorization.

3. EFFICIENT SCHEDULING MECHANISMS TO IMPROVE SIMD UTILIZATION

In the previous chapter, we successfully transformed a serial tree-traversal algorithm into a vectorized tree traversal algorithm using point-blocked code. This enables us to parallelize such serial trees using vector-processors, which are widely available on most architecture platforms. This is a good step and opens up opportunities to use available resources for faster execution. After the basic problem of generating work for vectorization has been addressed, the next natural question to answer is the effectiveness of this process. SIMD vectorization does not guarantee good utilization numbers and improved execution speeds all the time. When blocks of nodes execute in a vectorized manner, some nodes may “die out”, leaving the vector processors underutilized. Hence, the effectiveness of vectorization depends on many factors like memory layouts of data structures used in vectorization, fullness of the SIMD units, complexity of instructions executed, scheduling overheads and architecture-specific constraint for data transfer to the vector units. Among such factors, fullness of SIMD units executed depends on the compiler’s scheduler algorithms. For this scheduler to be effective, the utilization gain achieved by vectorization of the recursive trees must justify the overhead introduced to generate the vectors and their traversal for the remaining part of the tree. Since the scheduler can control the initial vector’s properties and the changes in its content and size during traversal, it plays an important role in optimization of vectorization. Hence, in this section, I will focus on how to improve the utilization of SIMD units for recursive trees by introducing new scheduling techniques to improve utilization, both by reorganizing existing parallel computations more effectively and by generating additional parallel work when necessary.

Ensuring that the blocks are almost always full requires stealing work from other parts of the tree where the nodes are untouched. As you can imagine, there is more than one way to do this. One way to do would be to randomly pick any node in the block , perform BFS of its children up to a certain depth and then add the resulting nodes to the current block. Another approach is to perform BFS on all its current nodes in the block till the block is full and then switch back to vectorized DFS. Yet another approach is to process another vector from the top of the tree till the current depth and add the new nodes to the old vector. These approaches are explained in detail in this chapter.

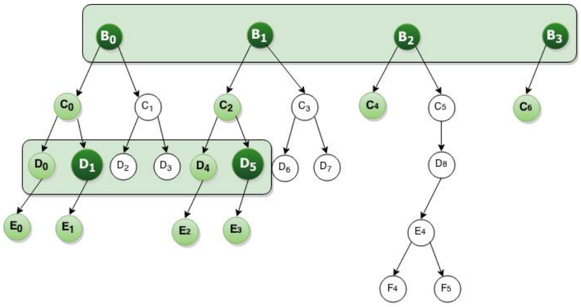
3.1 Single Step Re-Expansion(SSR)

3.1.1 DFS_#_# notation

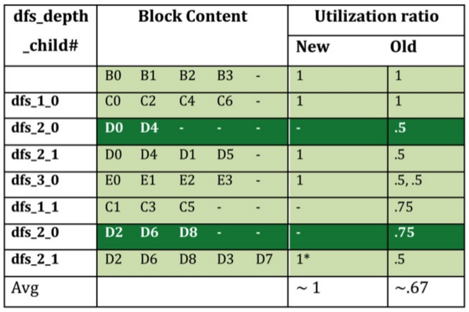
After BFS Work generation, the vector blocks get executed in a depth first manner. For the sake of convenience, the steps are named with a 'DFS' prefix in the figures below. The first number indicates the depth of the DFS tree traversal , assuming the 1st stage of DFS after BFS starts from a depth of zero. The next number indicates the iteration number of the block at the same depth. For example, in the table below, DFS_2_1 is the name of the block at depth 2 with both sibling nodes processed(0th and 1st). DFS_0_x represents a vector block at depth 0-assuming there exists a block of nodes ready to be processed, and all sibling nodes at the depth processed.

3.1.2 Policy explained

In this approach, we re-expand the SIMD stream by searching for more work from around the current node's parents. In short, for all the current nodes in the block, the scheduler looks for its unprocessed siblings and adds them to the block. This is illustrated with the tree traversal in 3.1(a). For all the policies, SIMD width of 4 is used as the default value. In table 3.1(b), the SIMD block status at each level is illustrated, following the same color codes as the tree traversal. For a block of size 4, assume that DFS_0_0 is the block available for vectorization after *work generation*.



(a) Single Step Re-expansion: Tree Traversal Pattern for a SIMD Block and its re-expansion to refill the block when its size shrinks is illustrated in this tree diagram.



dfs_depth _child#	Block Content					Utilization ratio	
						New	Old
	B0	B1	B2	B3	-	1	1
dfs_1_0	C0	C2	C4	C6	-	1	1
dfs_2_0	D0	D4	-	-	-	-	.5
dfs_2_1	D0	D4	D1	D5	-	1	.5
dfs_3_0	E0	E1	E2	E3	-	1	.5, .5
dfs_1_1	C1	C3	C5	-	-	-	.75
dfs_2_0	D2	D6	D8	-	-	-	.75
dfs_2_1	D2	D6	D8	D3	D7	1*	.5
Avg						~ 1	~.67

(b) Tree Traversal Table: Block contents and its utilization at each stage of SIMD vector operation is mapped in this table.

Fig. 3.1. Improvement in Utilization illustrated with Single Step Re-expansion

At dfs_1_0, the block has 4 nodes available and continues vectorization. However, in the next level, nodes C4 and C6 terminate and do not supply children nodes the block. Hence, the block size of dfs_2_0 reduces to 2. This results in a 50% utilization ratio. This impact is carried forward to the lower levels as well, resulting an overall utilization of 67%. Instead, at level 2, if we re-expand the block by addition the pending siblings , the utilization can be maintained. As highlighted in the table, re-expansion is triggered at both the dfs_2_0 stages after dfs_1_0 and dfs_1_1, causing better utilization.

Strip Mining

To avoid using vector operations when there is insufficient work, strip-mining technique is adopted. It is a degenerate version of loop tiling, where we have only one loop. In this technique, for a given block of nodes, vector operations occur in steps of SIMD Width. If there are few nodes left out after full-vector operations, they are executed in a serial fashion instead of vectors. This helps maintain utilization

```
// Block size = N, SIMD width = W
for (i = 0; i < N - W; i += W)
   SIMD_OP(); // Operates on floor(N/W) blocks as vector operations
  for (j = i; j < N; j++)
   OP();// Operates on the leftover N -floor(N/W) nodes serially.
```

Fig. 3.2. Strip Mining

while exploiting the advantage of interleaving serial and vector operation in SIMD hardware. The code in Figure 3.2 illustrates this. The second loop is a ”clean-up” loop in case W does not evenly divide N.

3.1.3 Scope of the Policy

Single Step Re-expansion works well only when there is sufficient work available around the block. Also, for sibling-blocks at the same level, re-expansion impact reduces from left to right. In the tree above, at dfs_x_1, re-expansion scope reduces. In general, work available reduces as one moves lower in the tree structure. Hence, while these techniques are effective, it cannot assure improved utilization for all tree structures. In the above tree, if C4 and C6 had a left child each, reexpansion would be triggered at a later stage, while processing a block with D1 and D5 . However, re-expansion fails since both D1 and D5 have no unprocessed siblings left. Hence, SSR alone cannot guarantee consistent utilization for all kinds of tree traversals. In the next scheduling policy, I extend the idea of SSR to multiple steps to ensure better performance.

3.2 Re-expansion - Switching between BFS and DFS Traversals

This second approach is something like a “mode swapping” BFS, where you have some set of points in your block, and you can either execute the points in that block in depth-first manner like how it is done in point blocking or in breadth-first manner

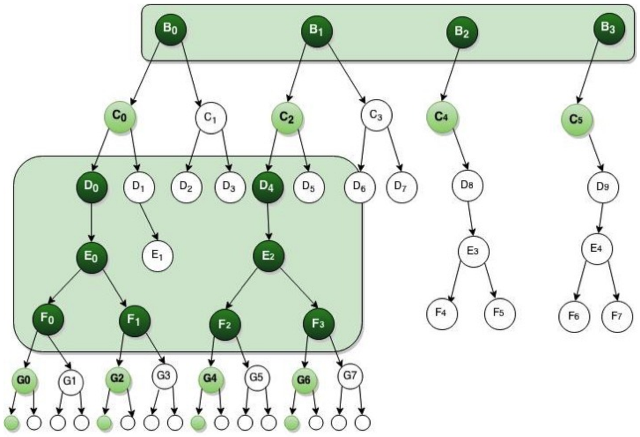
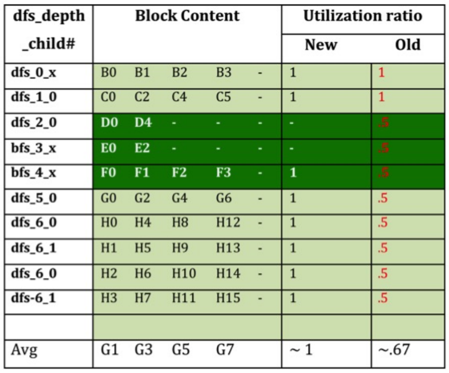


Fig. 3.3. Reexpansion Tree Traversal



dfs_depth _child#	Block Content	Utilization ratio	
		New	Old
dfs_0_x	B0 B1 B2 B3 -	1	1
dfs_1_0	C0 C2 C4 C5 -	1	1
dfs_2_0	D0 D4 - - -	-	.5
bfs_3_x	E0 E2 - - -	-	.5
bfs_4_x	F0 F1 F2 F3 -	1	.5
dfs_5_0	G0 G2 G4 G6 -	1	.5
dfs_6_0	H0 H4 H8 H12 -	1	.5
dfs_6_1	H1 H5 H9 H13 -	1	.5
dfs_6_0	H2 H6 H10 H14 -	1	.5
dfs-6_1	H3 H7 H11 H15 -	1	.5
Avg	G1 G3 G5 G7	~ 1	~.67

Fig. 3.4. Tree Traversal Table

(as in BFS expansion), and swapping back and forth switches the block between vectorized execution and work generation.

3.2.1 Policy Explained

In this technique, the scheduler does not depend on sibling nodes to generate more work. After the initial work generation stage, the block traverses in a depth-first-fashion till the minimum threshold for block size if encountered. At this point, the block of nodes switches its execution mode to breath-first expansion again to refill the block. All the nodes in the block execute BFS in a lock-step manner without using the SIMD vectors . Hence, the utilization ratio of the SIMD units is unaffected in this stage. When sufficiently filled up, execution switches back to Depth first traversal.

As shown in the tree traversal table 3.4, dfs_1_0 step has a full block and gets vectorized. However, at dfs_2_0, with only 2 nodes, the block is half-empty. Hence, the nodes D0 and D4 traverse in a breath first manner till the block size reaches at least four. This requires two levels for the given tree. At this stage, the block is full again and switches to depth-first traversal again. Using this technique, the utilization ratio for the steps traced in this traversal is 100%.

3.2.2 Scope of the Policy

In this technique, if the current block of points don't have many children or they don't have "bushy" trees below them, the technique may not be effective. Work generation by breath-first expansion of children nodes in a non-SIMD operation. Multiple levels of such operation defeats the purpose of solving the main problem of "Vectorization of recursive trees". While this technique may work very well for a large subset of trees, it surely does not address the vectorization of stringy trees. Hence, this is a light-weight scheduling technique that works for many recursive functions, but does not guarantee to work for all of them. In cases where both the above policies are not effective, we adopt restart mechanisms, this is explained in the next section.

3.3 Restart Policy

The previous technique involved searching for work around the current block by re-expanding the working block of nodes. However, in some recursive functions, when there is not much work available down the tree, multiple re-expansion calls need to be triggered relative to the depth of the tree. In such cases, it may be efficient to look at the upper part of the tree to generate work. This idea is explored in the re-start technique explained below.

3.3.1 Policy Explained

In this technique, I ensure that the blocks are almost full, by stealing work from upper nodes. To steal nodes from the upper level, I choose the initial block obtained after the first step of breath-first expansion. If more work can be found in the upper parts of the tree, looking for it at the top-most level possible maximizes the possibility

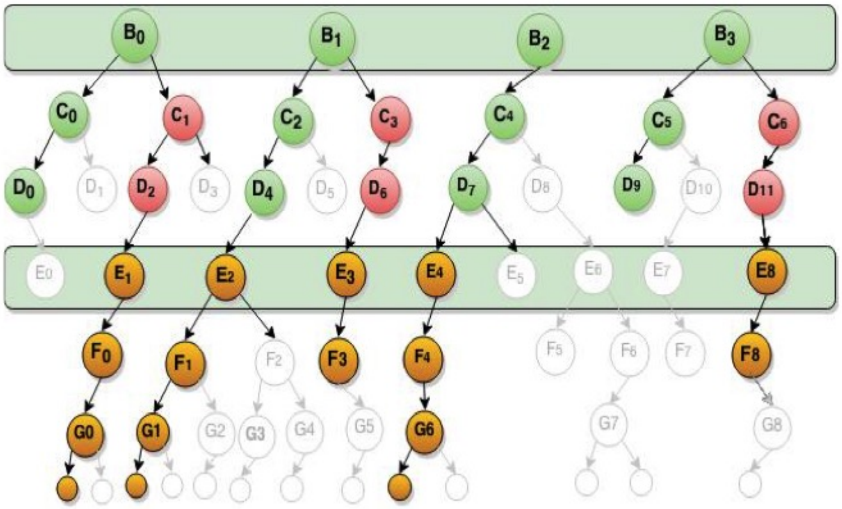
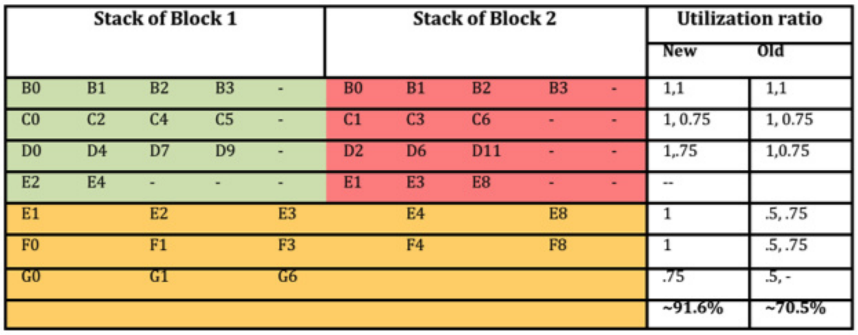


Fig. 3.5. Re-Start Tree Traversal

of generating more work. So, for re-start, we start operating on another un-processed block of nodes using the same techniques explained above. This can be imagined to be a new stack of blocks processed after the existing stack of block has halted execution at an arbitrary depth due to insufficient block size. When the new stack of blocks extends till the same depth where the old stack halted, the nodes from both the blocks coalesce to form a single new block. This block has continues to move down the tree as a new block.

In figure 3.5 below, the green nodes represent block-path traversed by the first block. When the block size drops , re-start function triggers creating and execution of a new stack of blocks . This is represented in red. At depth 3, the nodes from both green and red paths merge to form a new block of yellow nodes. This new block is wider than both the old blocks at the same level and provides better vectors to the SIMD vector. Hence, its utilization is guaranteed to improve.



Stack of Block 1					Stack of Block 2					Utilization ratio	
										New	Old
B0	B1	B2	B3	-	B0	B1	B2	B3	-	1,1	1,1
C0	C2	C4	C5	-	C1	C3	C6	-	-	1, 0.75	1, 0.75
D0	D4	D7	D9	-	D2	D6	D11	-	-	1,.75	1,0.75
E2	E4	-	-	-	E1	E3	E8	-	-	--	
E1		E2		E3		E4		E8		1	.5, .75
F0		F1		F3		F4		F8		1	.5, .75
G0		G1		G6						.75	.5, -
										~91.6%	~70.5%

Fig. 3.6. Tree Traversal Table

3.3.2 Scope of the Policy

This technique is very effective for extremely stringy trees, when re-expansion is not the most effective method of re-filling blocks. In-fact, this method is effective for any kind of tree- even the ones for which Re-expansion is efficient. However, it is not feasible to use re-start scheduling all the time because of the higher resources required for its implementation. This must be used only for functions where re-expansion fails. Scheduling for re-start requires more space and time and involves detailed bookkeeping of multiple stacks of blocks. Hence, this is memory and computation intensive in comparison to the previous scheduling policies.

Collectively, these 3 scheduling policies can be used with various recursive applications to vectorize them and maintain good utilization ratios. In the next section, I have tested the performance of these policies for a minimax function. However, since the T-Game restricts the shape of the recursion tree and does not result in fundamentally different trees with each run, it cannot be a good example for testing. Hence, I modify the T-game into a Generic Game and implement these policies over them. The game's implementation, its response to various scheduling policies and utilization graphs are explained in the next section.

3.3.3 Generic Minimax Game

In order to simulate SIMD utilization behaviour for different pattern of trees, we extend the idea of the tic-tac-toe minimax algorithm to a genetic game algorithm. For this generic game, we control tree structures by varying a set of parameters. These parameters are listed below.

Application Specific Parameters

1. Game termination probability(P) : Instead of following specific rules to determine if the given board state is won by a player or ended in a draw, a uniform distribution function is used to determine if a board terminates. Win or loss does not matter. Hence, any board state can terminate at any depth of the tree based on the value of P generated in the eval() function. This value is can be set to any value between 0 and 100. For example, a value of 30 indicates that the boards terminate about 30% of the times during evaluation at any stage of the game.

2. Boardsize (B): This indicates the length of the squared board used in the game. While the original game has a length of 3 (3 x 3 square) , the game to be extended to any N x N board by setting B=N. Higher values of B result in larger and deeper tree traversal and helps evaluation of SIMD performance. Architecture dependent parameters

3. SIMD-Width (W) : This indicates the size of the SIMD units used in the evaluation of vectorization. Usually, multiples of 2 are used as SIMD-sizes. This can be configured to emulate SIMD vector behavior of various applications with varying SIMD widths. Scheduler parameters

4. Block sizes : There are 2 block size parameters used in the experiments, namely max_blocksize and min_blocksize. These values are used for two reasons. The first reason is similar to the idea of block size in point-blocking. For a SIMD-Size

W, if the max_blocksize is N , floor(N/W) SIMD operations can occur either in parallel or serially. Hence, Hence, max_blocksize provides an upper limit on the number of SIMD operations that can operate in parallel or serially for a given tree-depth. However, when the number of nodes in the block drops below the SIMD width, utilization begins to take a big hit. At this stage, utilization improvement techniques like reexpansion and re-start must be triggered. Hence, min_blocksize is used to trigger these scheduling operations. Min_blocksize sets the minimum block size required for SIMD opearations.

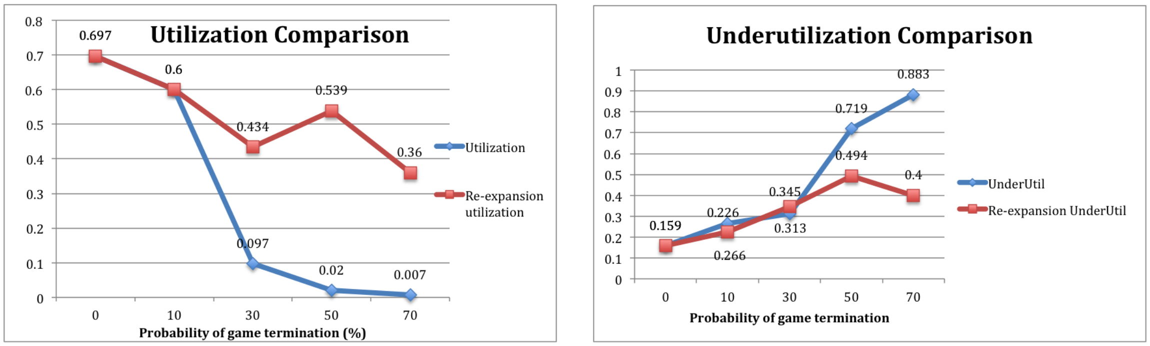
With these parameters, I tested the robustness of the restart and re-expansion policies for different kinds of tic-tac-toe kind of minmax applications. This expands the evaluation samples and provides a better estimate of the utilization benefits of the new scheduler.

3.3.4 Re-Expansion Scheduling Evaluation

Profiling of re-expansion scheduling for a generic 3x3 minmax game.

In order to improve the utilization of SIMD units, re-expansion can be done at different depths in the tree. Based on an application specific policy, when the number of nodes in a given block falls below a certain threshold, its parent block can be re-expanded to generate more work to fill the block. This can be done with minimalistic changes to the existing design and results in higher utilization ratio for bushy trees. The graphs in Figure 3.7 shows the improvement in Utilization and Underutilization numbers achieved by re-expansion. The performance of the minmax algorithm with re-expansion is compared with its older version that does not perform re-expansion. The assumptions and parameter values used for profiling are noted below.

1. To show improvement in numbers, I have profiled for SIMD width of 128. Since the work generated by a 3x3 minmax game quickly dwindles to values less than 128, the role of re-expansion can be observed clearly.



(a) a

(b) b

Fig. 3.7. Impact of Re-expansion on SIMD Vectorization

2. All the points available after *Work Generation* are considered to form a single block for vectorization.

3. The threshold limit that triggers re-expansion is set to be 1/3rd the block size. This is a policy parameter and will depend on the application.

The graph in Figure 3.7 shows that as the probability of game termination increases, the re-expansion code performs generated more work and hence results in better utilization and underutilization numbers. For probability values lesser than 30%, the re-expansion function will not be triggered sufficient number of times since the blocks are fairly full. Hence, SIMD performance does not vary too much. As the probability increases, nodes are dropped from the block more frequently. Hence, reexpansion re-fills the blocks and results in improved utilization.

Variation in Utilization for different SIMD widths

The graph in Figure 3.8 shows the changes in performance of vectorization for variation in SIMD width. As seen in the figure, utilization improves with SIMD width up to a certain point and then drops. An important point to consider while analysing this graph is that the maximum size of the block on which point blocking can be applied for a 3x3 board with BFS depth of 3 is 336. Hence, for SIMD widths

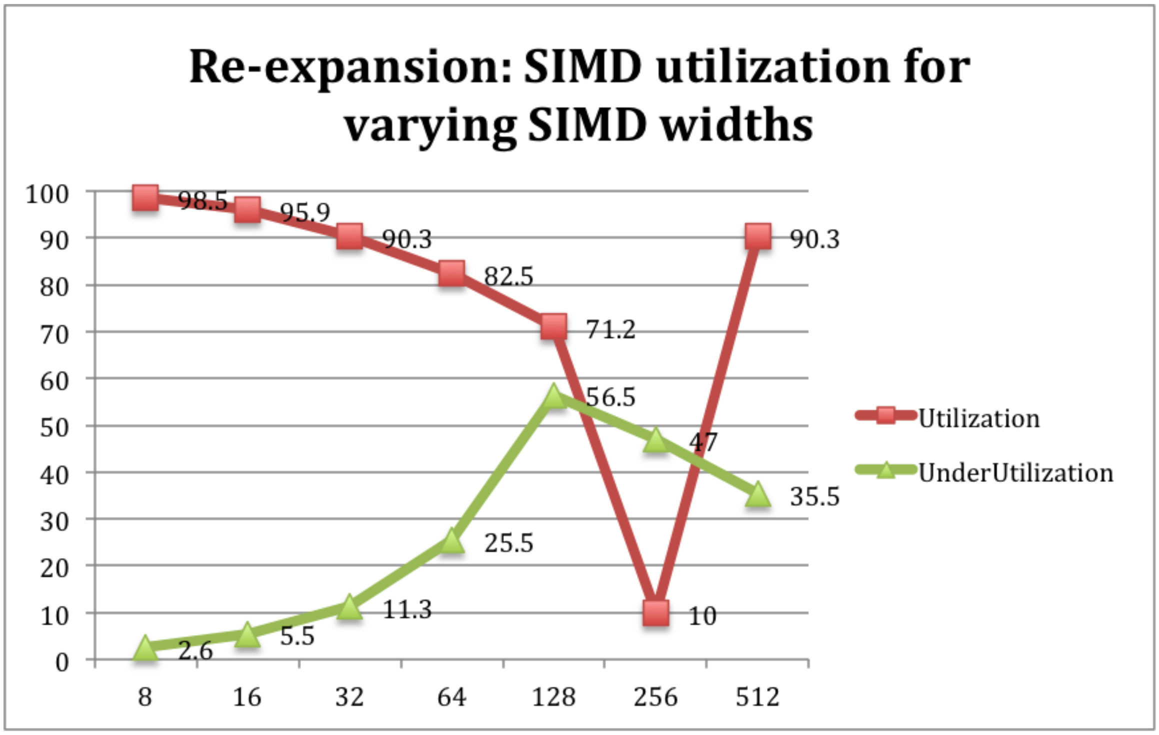


Fig. 3.8. Impact of SIMD width variation on Vectorization

from 8 to 256, the utilization decreases gradually. However, when the width is 512, all the points fit into a single block. Utilization depends on the number of empty spots in the block henceforth. This explains why it shows better utilization in comparison with 256.

Variation in Performance for different re-expansion thresholds.

The graph in figure 3.9 provides insight into minimum operable block sizes to guarantee good utilization. Threshold values here refer to the minimum fraction of the full block size that is required, for vectorization to continue. When the block size drops below this fraction, reexpansion or rescheduling is triggered.

As seen in the graph, for threshold values below 50% of the block-size, the utilization improves as the fraction increases. This is because, if we consider the algorithm's

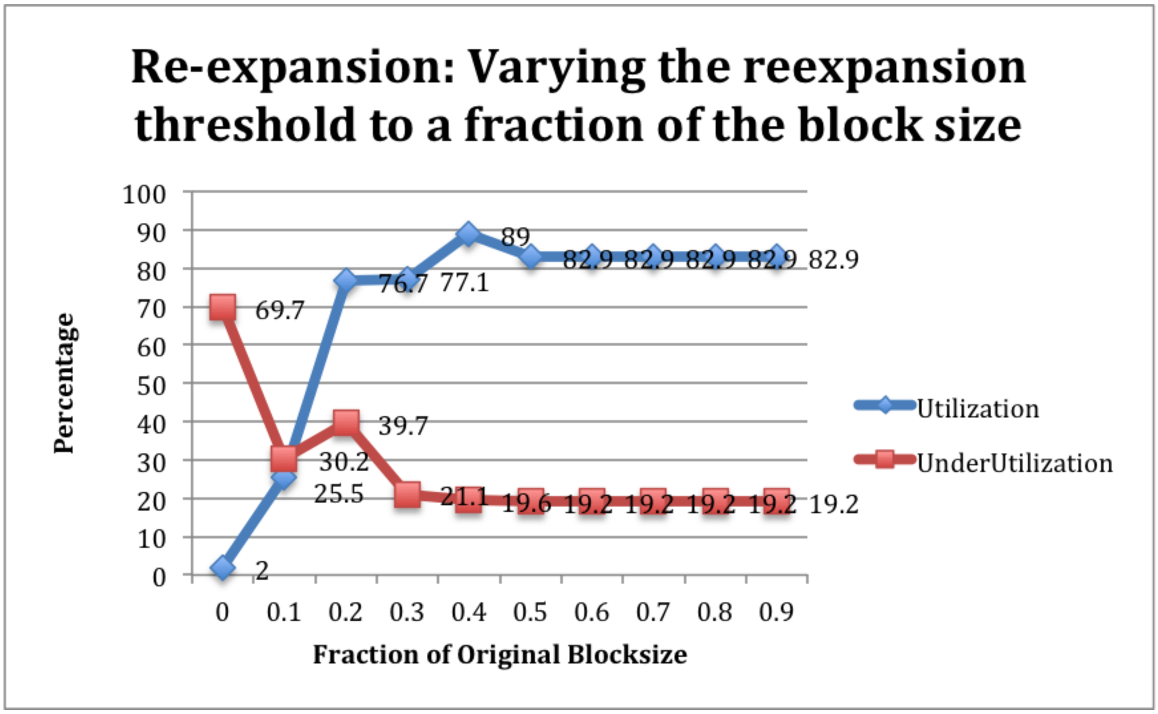


Fig. 3.9. Impact of re-expansion threshold point variation on SIMD Vectorization

timeline, re-expansion is invoked at an earlier time (higher levels of the tree) for blocks with higher thresholds and much later(at lower levels of the tree) for lower values . Hence, better utilization is seen. However, when the threshold greater than 50%, reexpansion is not useful. Why? Because re-expansion tries to add multiple siblings for each node in the block. Hence, the size of the reexpanded block gets multiplied by the number of siblings added per node. So, to have reexpansion, a minimum of one sibling from each node in the block is necessary. If the number of nodes in the block greater than 50% of block size, the reexpanded block is bound to overflow. Hence, re-expansion is not performed. This can be changed such that, we allow the block to overflow and handle the spill as a separate block. But this is not done in the present implementation.

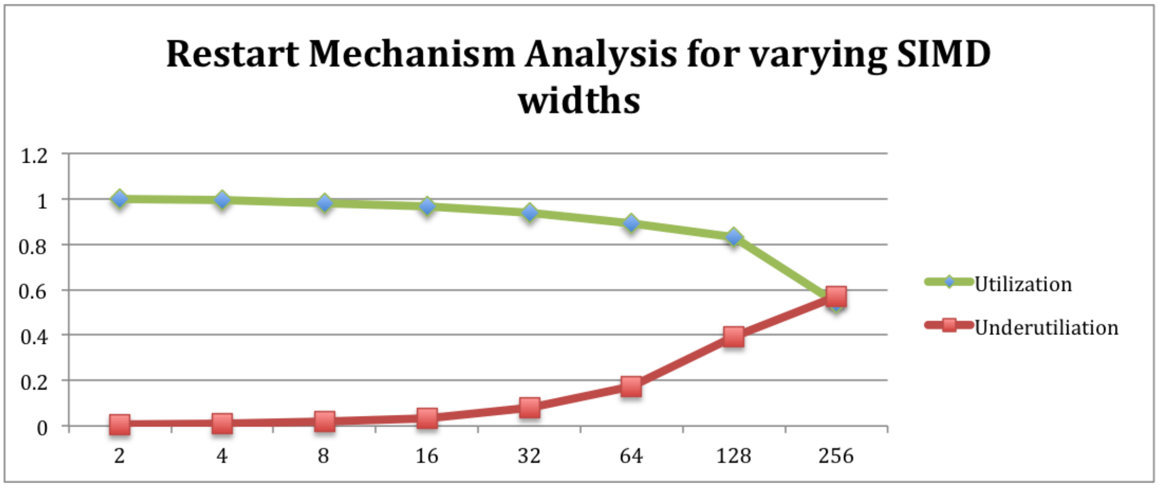


Fig. 3.10. Impact of SIMD width variation on Vectorization

3.3.5 Restart Scheduling Evaluation

Since Re-start mechanism is designed to address recursive cases where re-expansion fails, it is important to analyse the fullness of tree structures as well. When the tree fullness is low, re-expansion performs poorly. In such cases, restart works better. Hence, to analyse restart mechanisms, we perform profiling similar to reexpansion and then, compare the two scheduling policies for the same functions.

Graphs in Figure 3.10 and 3.11 show the performance of restart mechanism for the generic minimax algorithm. Similar to Re-expansion, Utilization and Under-Utilization improvements seen is shown in the plots in 3.10 and 3.11. Performance improvement is similar to that of re-expansion. However, we modify the tree behavior here by changing the probability of termination of each board sample of the game. Higher termination probability results is less-full trees. For such trees, re-start's impact is seen prominently.

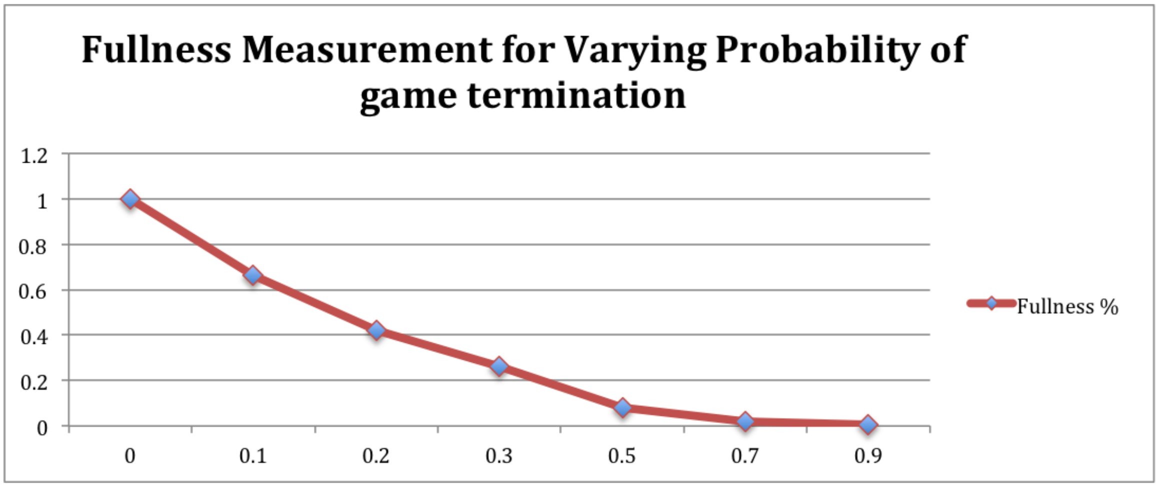
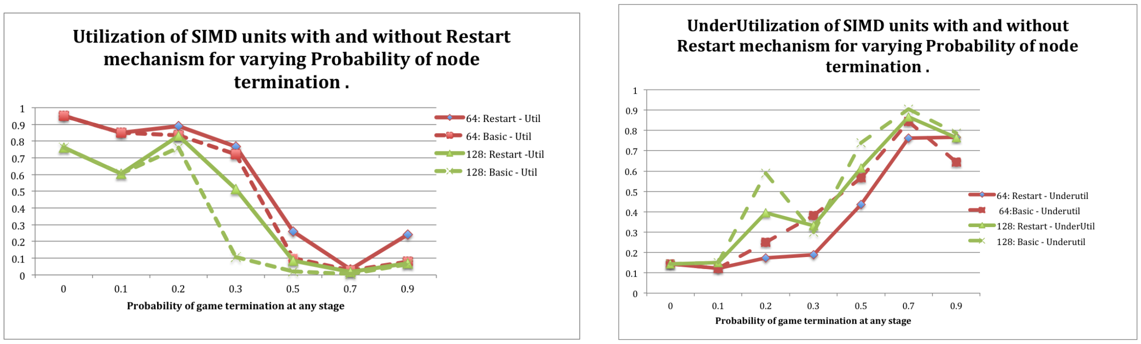


Fig. 3.11. Variation of fullness with probability of the game

Performance of re-start scheduling policy

The restart mechanism's performance is measured in terms of SIMD utilization and underutilization and compared with the basic application that does not support restart/reexpansion. These graphs vary depending on SIMD width and number of times restart is triggered. The following graphs compare the performance for SIMD Width of 64 and 128.



(a) a

(b) b

Fig. 3.12. Impact of Re-Start on SIMD Performance

The graphs show us that restarting helps improve the performance of SIMD vectorization as expected. The deviation from the basic implementation is highly dependent on the SIMD width, tree structure and number of restarts that can be invoked. In these examples, I limit the number of restarts to 2 per turn of the game. The table to the left further emphasizes the fact that the level at which restart is triggered increases as the probability of termination decreases. Hence, irrespective of the current state of the block, restart can be triggered when work must be generated.

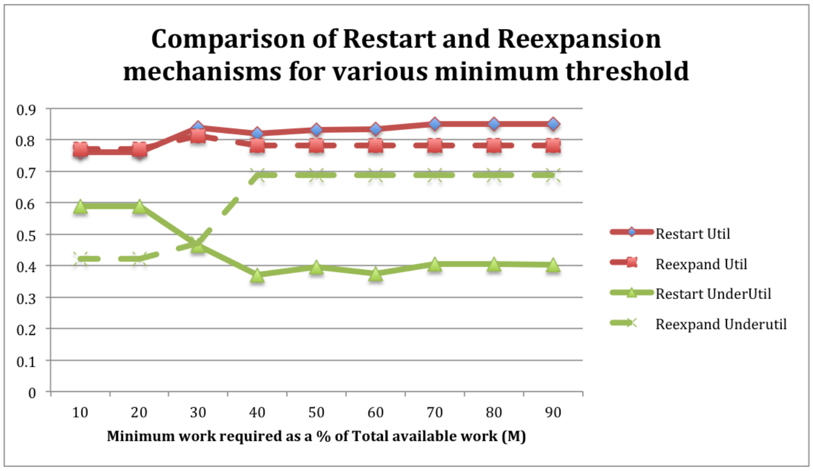
Probability	**RestartDepths**
.2	6,7
.3	5,6
.5	5,5
.7	4,4

Table 3.1.
Tree Shapes and Re-Start Trigger levels

As seen from the table, as the termination probability decreases, the trees become less bushy. This results in re-start functions getting invoked higher up in the tree in comparison the fuller trees.

3.3.6 Comparison of ReStart and ReExpansion scheduler policies

In this graph, I have compared the effectiveness of re-expansion and restart mechanisms. Note that re-expansion and restart are triggered by the same minimum conditions. Re-expansion can occur any number of times, as long as there is work available at the sibling nodes of a given block. However, restart can occur only once for every child node of the BFS block. Effectively, for a 3x3 minmax algorithm with BFS depth of 3, restart can be triggered not more than 3 times. The graph shows that restart gives better utilization almost all the time. However, Underutilization ratio of re-start is much better than that for re-expansion, more so for higher values



M(%)	A	B
10	0	0
20	0	0
30	2	17
40	2	54
50	2	54
60	2	54
70	2	54
80	2	54
90	2	54

Fig. 3.13. Comparison of the behavior and impact of both the scheduling policies- Reexpansion and Restart. M = % of block size that triggers new work generation. A and B represent the number of times Restart and Reexpansion policies are triggered during the execution of the program.

of M. This is because, for M greater than 50%, we cannot perform re-expansion since the mechanism will generate more work than what the block can accommodate. In the case of re-start, since a new stack is allocated for each restart call, this problem will not arise.

The table here compares the number of restart and re-expansion calls that are made for 1 turn of the game. Clearly, a single restart function is more powerful than multiple re-expansion functions when SIMD widths are chosen wisely. The pros and cons of both the policies are already explained in the beginning sections of the chapter. Hence, based on the tree traversal, one of these scheduling policies can be applied.

3.4 Conclusion

This chapter explain the scheduling polices and showed its impact on the 3x3 TicTacToe game. Now, the scheduling policies' behavior for other recursive trees must be analysed. The next chapter explores the various possible tree traversal shapes that out scheduler can vectorize and profiles its performance for the same. It supports the results shown in this chapter and consolidates our observations.

4. SIMD VECTORIZATION OF RANDOMLY SYNTHESIZED GENERIC TREES

The evaluation done so far has been on tic-tac-toe based minmax algorithms. The effect of SIMD width, block size and work generation techniques for such functions are explained in the previous chapter and provide a sound idea about architecture parameters. However, this does not provide information about the behavior of our scheduler for various kinds of trees. For example, the minmax algorithm's probability function for evaluation of board termination creates similar kinds of subtrees.It does not generate trees where some parts are very bushy and some are very stringy. Hence, in order to understand the behavior of the scheduler for all kinds of trees, we generate random trees and evaluate SIMD behavior and its utilization.

4.1 Tree Topology

Tree building is enabled by controlling the shape of the tree using a few parameters. Mainly, the tree structure is controlled by two parameters.

1. Number of nodes in the tree(N)
2. Maximum Depth of the tree(D)

N nodes get distributed randomly to form a rooted binary tree where there is at least one path that reaches depth D. The main idea of providing these 2 parameters is to enable building trees that will aid in measuring the effectiveness or limitations of our vectorization algorithms designed so far.

4.2 Random Tree Generation

To build a binary tree with nodes **N** and depth **D**, a standard recursive binary tree building algorithm is used. The criterion to meet are that the tree depth must not be greater than or lesser than D, while every run of the program produced a random tree. The idea here is to profile vectorization for binary trees with unknown tree shapes and prove its effectiveness for any binary tree. In order to build a tree based on random tree splits one must ensure the tree depth is not more than D at any time.The pseudocode for tree building is shown in Figure 4.1. The tree structure generated using this code is used as the input for vectorization and analysis of various work generation policies.

4.2.1 Implementation

In the code in Figure 4.1, the paths at each split are labelled *red* and *green.* The green path must traverse till the maximum depth 'D'. The red paths can terminate at any point before till depth 'D'. To build the tree, I ensure that the following conditions are met at each split.

1. For the Green path, the number of nodes in the subtree should be sufficient to at least form a single stringy path till 'D'. The maximum nodes permitted is power(2,depth of remaining sub tree). Hence, in the code, *minNodes* and *maxNodes* provide the min and maximum node conditions for the green path.

2. For the red path, the maximum number of permitted nodes is limited by the maximum depth of the subtree possible. The tree cannot go beyond depth 'D'.

3. The split is a uniform random number between one and total nodes available at the subtree root.

In this design the partitioning ratio at each level is determined randomly, in a unifrom fashion, based on the total availability of nodes. A randomly generated valid number

```
max_depth D;
void buildtree(nodes n, depth d, path color)
  if(d == D)
    return 0;

  while((n>1)) //Nodes available for tree-building
    minNodes  = D-d;  //Min nodes needed to reach depth D
    maxNodes =  pow(2,D-d);//Max nodes that can fit into the subtree
    path (lcolor,rcolor)=RED
      if(GREEN ==   color)//This subtree must traverse till depth 'D'
      split = rand() mod (n-1);
      if( (split >= max(minNodes, n - maxNodes))&&
          (split <= maxNodes) )
          lcolor = GREEN;
          break;//Valid split
      //else, loop again and try another split
    else if(RED == color)
      split = rand() mod (n-1);
      if( (split <= maxNodes)&& //Ensure these trees don't go below 'D'
      (n-split <= maxNodes))
        break;

  buildtree(split,d+1,lcolor);
  buildtree(n-split,d+1,rcolor);

int build(nodes n)
  buildtree(n,0,GREEN);
```

Fig. 4.1. Binary tree table algorithm

between 1 and the total number of child nodes available provides the split ratio for branching a node into 2 children nodes. This process continues for every node split at every level of the tree. A tree built with such an algorithm provides a fairly unique

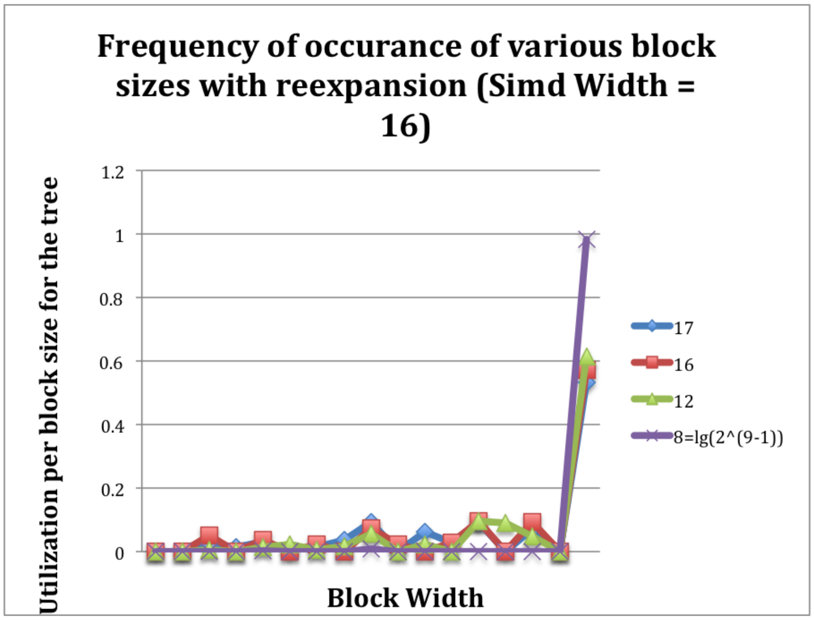


Fig. 4.2. Utilization Distribution among various SIMD states for 4 groups of tree-topologies

tree for each run of the program with the same parameters depending on the total number of tree forms that can be theoretically built.

4.2.2 Performance Analysis

With the above implementation, I profiled vectorization with *re-expansion* technique for a number of tree structures. The results of these experiments show that re-expansion ensures a good utilization ratio for SIMD operations on recursive trees.

In Figure 4.2, the plot sheds light into analysis of SIMD utilization based on three important parameters. The parameters used in this experiment are - SIMDWidth(W) is 16, Maximum width of the block(Max) is 64, Minimum block size to trigger re-expansion(Min) is 16, number of nodes in the tree is 511(N).

1. Depth of the tree

   In order to test for various topologies of trees, the graph tracks the utilization for 4 different depths of tree. In this evaluation, for 511 nodes, a full binary

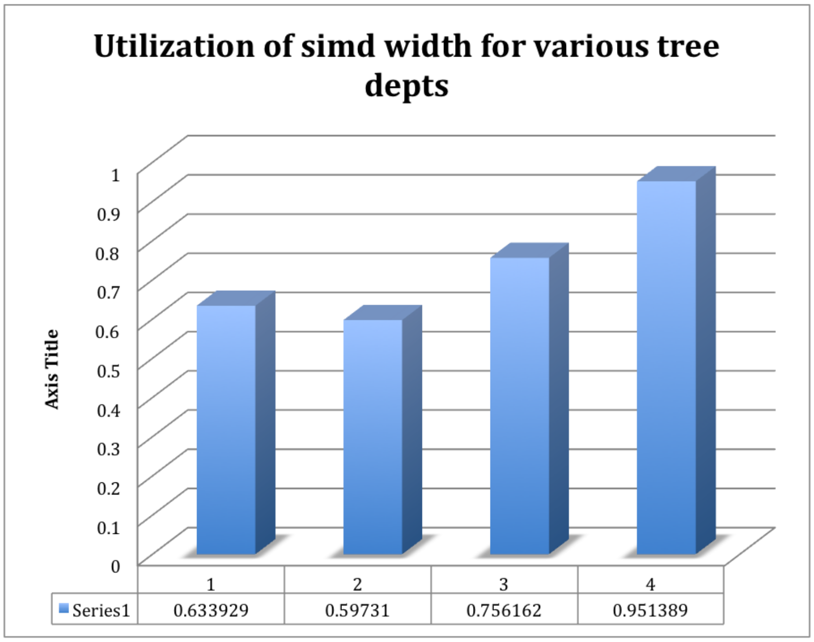


Fig. 4.3. SIMD Utilization for 4 groups of tree-topologies

tree has a depth of 8. In these experiments, the focus is to see the behavior of the scheduler for constant * full-tree-depth. In this case, we profile for depths of 8, 12, 16 and 17.

2. SIMD States

   Re-expansion is triggered only when the block size goes below Min. However, for larger blocks, they get vectorized in batched of size W. Hence, the last part of the block can still get execute with under-utilization. State of the SIMD Block is defined by how full the unit is. In the graph, the distribution of work among various states of hardware is profiled. 100% of the work does not happen at width 16.Some of it gets distributed in partial SIMD units as well. The X-axis of the plot if Figure 4.2 represents the width of the SIMD units while vectorizing blocks in the algorithm.

3. Minimum and Maximum Block sizes

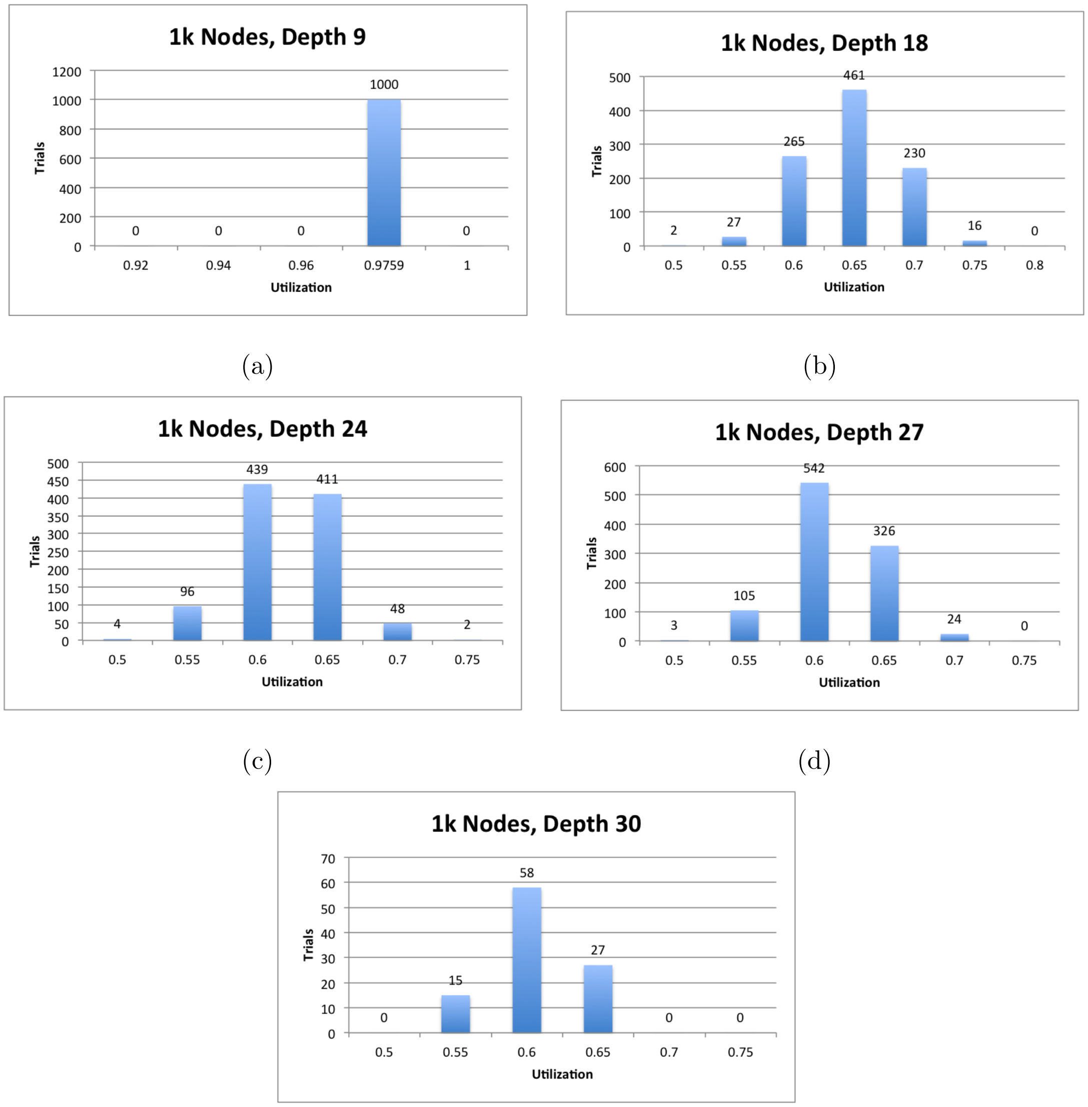
   This is a heuristic value based on the algorithm. It is set based on the how well or badly the scheduler performs for a given combination of parameters.

4.2.3 Repeated Trials

The graph in Figure 4.2 shows that the utilization does not degrade below 50% in most cases. The values are profiles by averaging the result of running each experiment 1000 times. Figure 4.3 shows the overall average Utilization values for trees of depth 17,16,12 and 8 for the same set of parameters mentioned above. The average utilization stays above 50% for all the trees groups profiled. This is a significant result as it helps guarantee a lower limit of the utilization gain obtained through vectorization.

Since these profiling data were averaged over 1000 trials, the variation in utilization during each trial is sorted into bins in Figure 4.4. Figure 4.4(a) represents the utilization for a fully balanced binary tree. As expected, each trial produced the exact same number, 97.59% since they all operated on the same tree. For the rest of the graphs, each trial run generated a different topology of the tree for the same set of nodes and maxDepth. The variation in performance between these trials is represented in the graphs. Clearly, for all the sample runs, bulk of the trials had utilizations greater than 50%.

Now that we have sufficient data to support that re-expansion is almost always effective for upto 3*fulldepth depth trees, there is another subtle factor to be considered to strengthen our argument-which is the random tree generation. The trees generated in these trials are not true-uniformly random samples. The next section analysis the sampling policy and explains details about the implementation of a more accurate node-splitting policy.



(a)

(b)

(c)

(d)

(e)

Fig. 4.4. Utilization bins for various tree-topology groups over 1000 trials

4.3 Uniform Sampling of Node Splits

The nodes available at each stage are split based on the random number generator in the previous analysis.If *split* in Figure 4.1 is a random variable between 1 and total nodes, the cumulative distribution function of *split* is uniform. However, binary tree building algorithm does not follow a uniform distribution. It is important that we sample a truly random tree-space to guarantee the behavior of our efficient vectorization policies. In order to achieve this, I have written a function that stores the number of possible binary rooted trees that can be constructed given the nodes and depth of the tree. This function is based on the idea of catalan numbers and stores the result in a table. The table is used to create splits in the tree building algorithm. The code for this function is shown in Fig 4.5. Here, the array *trees_exact[]* is used as a reference to generate a probability distribution function for binary tree sampling. The nodes at any stage are then split by referring to this PDF instead of a random number generator. Hence, a new split_finder() function replaces the existing *rand()* function in line 10 of the code in Fig 4.1.

4.3.1 Generation of Probability Distribution for Binary Trees

Given 'N' nodes, the total number of binary trees that can be constructed using these N nodes is given by the closed form series of Catalan numbers. A binary tree used in our experiments has the condition that each node in the tree is either a leaf node or has two children. In other words, no node has an odd number of children. Hence, all internal nodes have 2 branches. However, this does not take into account the depth restriction required for our tree-building problem. Hence, to count trees given (N,D), we use a bottom-up approach explained below. Given the number of trees with 'N' nodes and depth D, we can calculate the trees with node 'N'+2 as follows. Put the N+2nd node as the root. Put the subtree with N nodes as the left subtree and the N+1th node as the right subtree. Count the total trees with this combination. Next, split the nodes into (N-2, 3) split and count the total trees. Continue this till a split of (1, N) is encountered. The cumulative sum of all these splits gives the total value. Hence, we can formulate an initial *trees_table[]* and use dynamic programming approach to extend the table. The pseudo-code for this is as shown in Fig 4.5.

```
long eval_treenum(nodes n, depth d)
  /*All split-combinations from 1:(n-1) to (n-1):1*/
  left = 1, right = n-1;
  for each(node i : n-1)
      a = tree_total[left][d-1]
      b = tree_total[right][d-1]
      /*For each of the 'a' tree shapes on the left,
        'b' right trees are possible*/
      total += a*b
      left++;
      right--;
  return total;

int build_tree_table(nodes N, depth D)
    for each(nodes n: N)
      for each(depth d:D)
      /*Evaluate all trees possible with nodes 'n' and maximum depth 'd'*/
        trees_total[n][d] = eval_treenum(n,d)
    /*To get the exact trees with depth 'd' and nodes 'n',
     eliminate all trees with depth < 'd' and nodes 'n'*/
    trees_exact[n][d] = trees_total[n][d] - trees_total[n][d-1] ;
    return true;
```

Fig. 4.5. Binary tree table generation

Implementation

The algorithm follows a dynamic programming approach to determine the maximum number of trees possible with nodes N and maximum Depth D. To enable this, we initialize the 2-D tree table *trees_total* with the arrangement shown in Table 4.1. This information stored in the *tree_exact* array forms the probability space for uniform sampling of all possible tree shapes , given the node and depth constraints. Hence,

2	0	-	..	-
1	1	1	..	1
0	0	0	..	0
Node#	**0**	**1**	..	**D**
	Depth			

Table 4.1.
Initialized Table for Tree Count

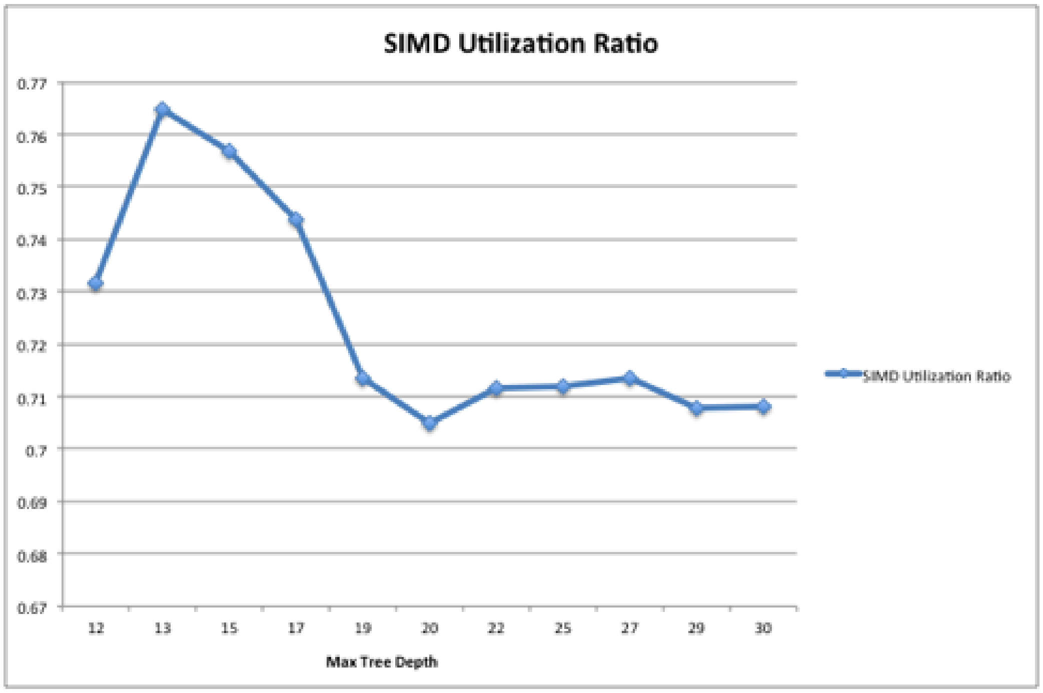


Fig. 4.6. Average SIMD Utilization of randomly generated rooted-binary trees

the random sampling function used to generate a binary split at each level of the tree is now truly uniform.

4.3.2 Results

To reverify the results of vectorization with the new sampling, we profile vectorization with the new tree building function for 2k nodes and SIMD width of 16. For 2k nodes, a full binary tree has a depth of log(2k)= 10. The idea here is to test the scheduler's behavior for trees with depths based on the 2 rules below.

1. A small constant added to the full_tree_depth

2. Multiples of full_tree_depth .

The graph in Fig 4.6 tracks the utilization ratio for various trees with 2k(2047) nodes and SIMD width of 16. The behaviour of re-balancing scheduler is analysed here for trees whose depth varies from 12 to 30. As seen in the graph, the utilization ratio is steady at around 70% and does not drop drastically with increase in tree depths.

4.4 Profiling-based Evaluation

We now evaluate the performance of our vectorization techniques using the tree generation technique outlined in the previous section. Re-expansion exploits SIMD hardware by creation and maintenance of data blocks. Therefore, SIMD width, block size, re-expansion thresholds, and tree shapes (Nodes and Depth) determine the benefits from SIMD utilization. Individually, these parameters show their distinct impact on vectorization. Together, they are useful to analyze vectorization behavior in systems where there are constraints on some of these parameters.

For our analysis, we simulate vectorization with the following parameters as default values, unless otherwise specified: SIMD width of 16, block size of 64, and 10,000 nodes in the tree. All values are averaged over 100,000 trials. We perform our simulations on an system with a 2.6GHz 8-core Intel E5-2670 CPU with 32KB L1 cache per core and 20MB last-level cache. The simulation code was compiled with Intel icc-13.3.163 compiler with '-O3'.

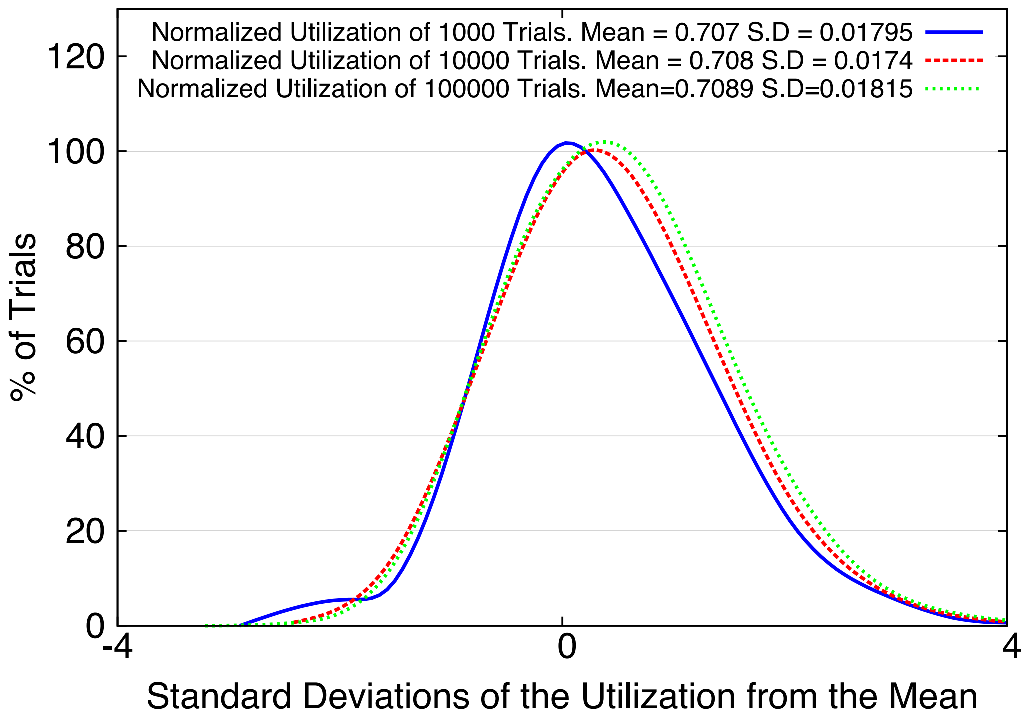


Fig. 4.7. Central limit theorem used to show that for 1k, 10k and 100k trials, the distribution of utilization values is fairly normal.

4.4.1 Repeated trials to sample utilization

We measure average utilization over repeated independent trials to get consistent results. Because all these trials are independent, sufficiently large trials will comply to the central limit theorem. Using the default system settings of SIMD width 16, block size 64, and 10,000 nodes, we choose a tree of depth 20 to show that the variation in utilization over 100,000 trials obeys the theorem (Figure 4.7). As seen in Figure 4.7, the distribution of trial values follows a normal distribution and remains consistent for 1000, 10,000 and 100,000 trials. Based on this observation, we perform all experiments with 100,000 or more trials to produce consistent averages.

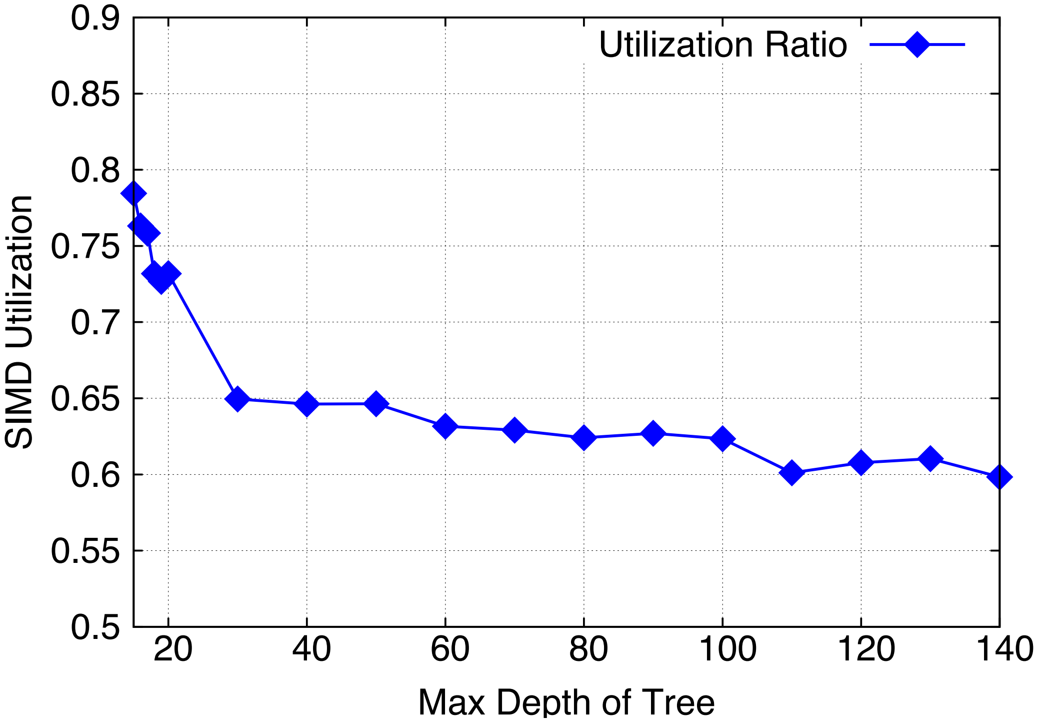


Fig. 4.8. Utilization ratio remains well above 50% for all tree depths

4.4.2 SIMD utilization for deep trees

We empirically evaluate the SIMD utilization of the depth-first and re-expansion strategies for trees whose depths are multiples of full-tree depth. SIMD utilization ratio is high for full trees. As tree depth increases, the trees can get more sparsely populated. Maintaining high SIMD utilization for such trees can become difficult. Re-expansion addresses this by switching between vectorization and work generation. To see its impact on SIMD utilization in the context of deep trees, we profile vectorization behavior for a 10,000-node (full-depth of 13) tree with varying depths. We start with depths around the full-tree depth and evaluate depths that are multiples of full-depth. As shown in the Figure 4.8, for trees with depths varying from 14 to 150, utilization stays above 50%.

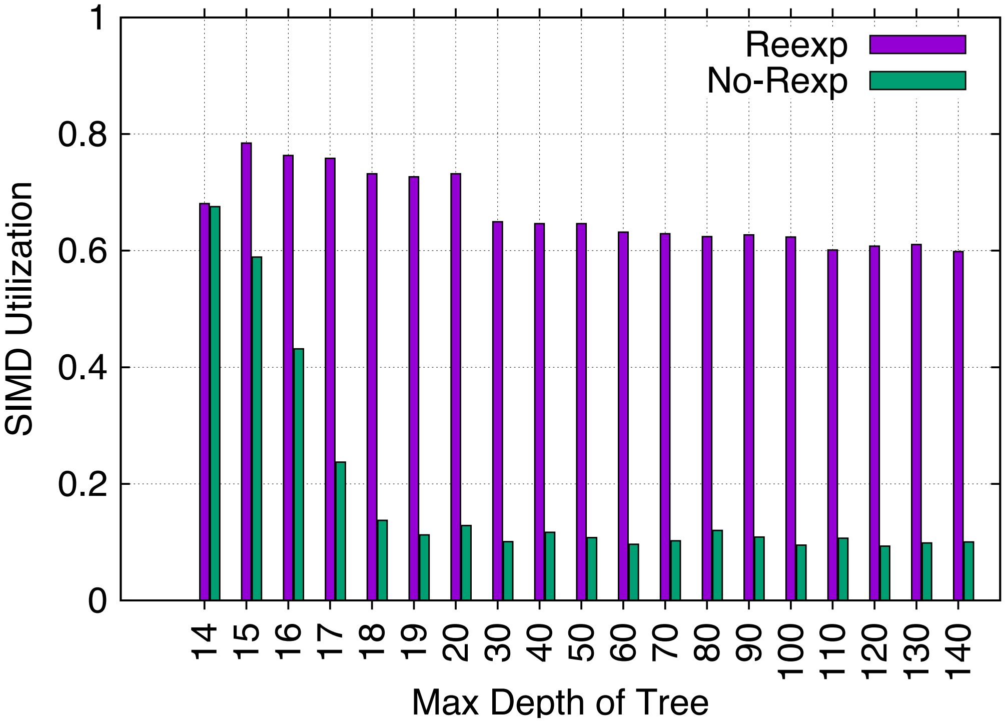


Fig. 4.9. Impact of Re-expansion: Utilization achieved with and without re-expansion for various class of trees

4.4.3 Impact of re-expansion

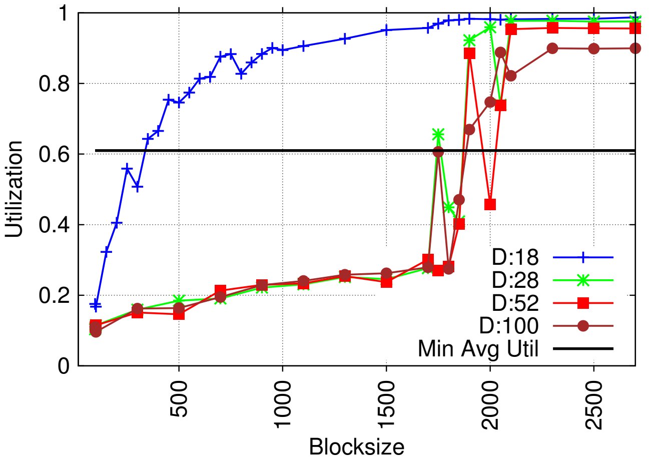
We measure the impact of re-expansion by comparing vectorization efficiency with and without re-expansion. Further, we analyze this impact for various class of trees to emphasize its significance. For 10,000-node trees, we have an almost full-tree at depth 13. In Figure 4.9, vectorization for random trees between depth 14 and 140 is measured with and without re-expansion (for a fixed block size). For depth 14, plenty of work is available and re-expansion does not significantly improve utilization. For the same number of nodes, as the trees get deeper, the sparse nature of the trees expose inefficiencies incurred by depth-first execution. For depths greater than 20, re-expansion improves vectorization by ∼5 times.

We observe that re-expansion improves utilization for a given system configuration. To attain similar performances without re-expansion, depth-first scheduling without re-expansion needs to generate more parallel work at the cost of memory by using larger block sizes. Large blocks perform breadth-first execution till deeper levels in the tree to fill the chosen block size.

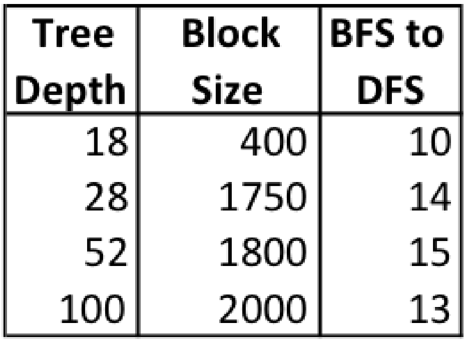
An alternate way of demonstrating the advantages of re-expansion is to study how re-expansion allows us to use significantly smaller block sizes, and hence consume less memory. We choose 4 trees of depth 18, 28, 52, and 100 as representative depths. When run *with* re-expansion, and a block size of 64, the average utilization achieved for these depths was 76%, 66%, 65%, and 61%, respectively. We then turned off re-expansion, and studied how utilization changed with block size. The results are in Figure 4.10(a). As in the previous study, we see that at a block size of 64, the non-re-expansion runs have poor utilization. More significantly, we see that to achieve the same utilization as re-expansion with a block size of 64, the non-re-expansion runs require block sizes of over 400, 1750, 1800, and 2000, respectively. We also see that such large block sizes mean that significant chunks of the tree must be explored before switching to depth-first execution, as seen in Figure 4.10(b). These results emphasize two points: (1) to match SIMD utilization without re-expansion, block size needs to significantly increase with tree's depth and (2) the utilization achieved is less predictable.

Variation in SIMD width

SIMD width and block size together decide the percentage of work that gets vectorized. Block size is usually greater than the SIMD width and holds multiple vectors for efficient vectorization. Keeping the SIMD width a constant, the variation in utilization with changing sizes of the block is shown in Figure 4.11. As shown in the figure, increase in block size relative to SIMD width causes better packing of data for vectorization. For block size equal to SIMD width, utilization is poor and drops to 40% for deeper trees. Bigger blocks result in lesser re-expansion calls and higher percentage of execution performed by vector units.



(a) BlockSize Increase to achieve higher utilization ratio



Tree Depth	Block Size	BFS to DFS
18	400	10
28	1750	14
52	1800	15
100	2000	13

(b) Shows the depth at which vector operations begin for various trees classes

Fig. 4.10. Utilization vs. block size for different tree depths without re-expansion. The horizontal line represents the minimum average utilization when using re-expansion with a block size of 64 for the same tree depths.

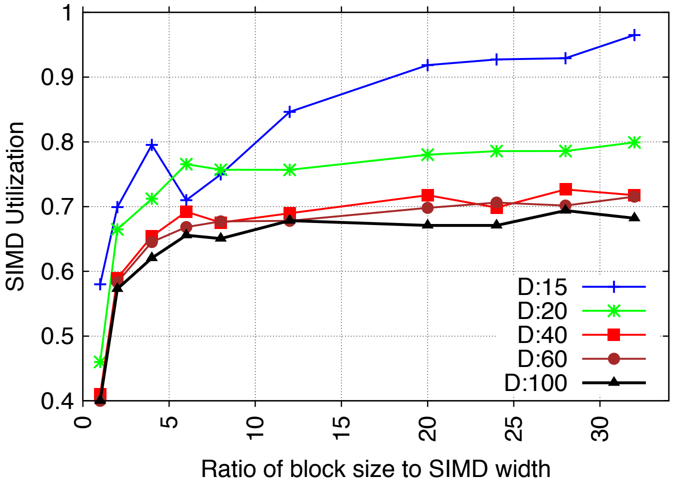


Fig. 4.11. SIMD utilization vs ratio of block size to SIMD width. As the radio increases, amount of work available for each SIMD operation increases. Hence, a greater percentage of full-vector operations occur as compared to smaller block sizes. This ensure high average utilization.

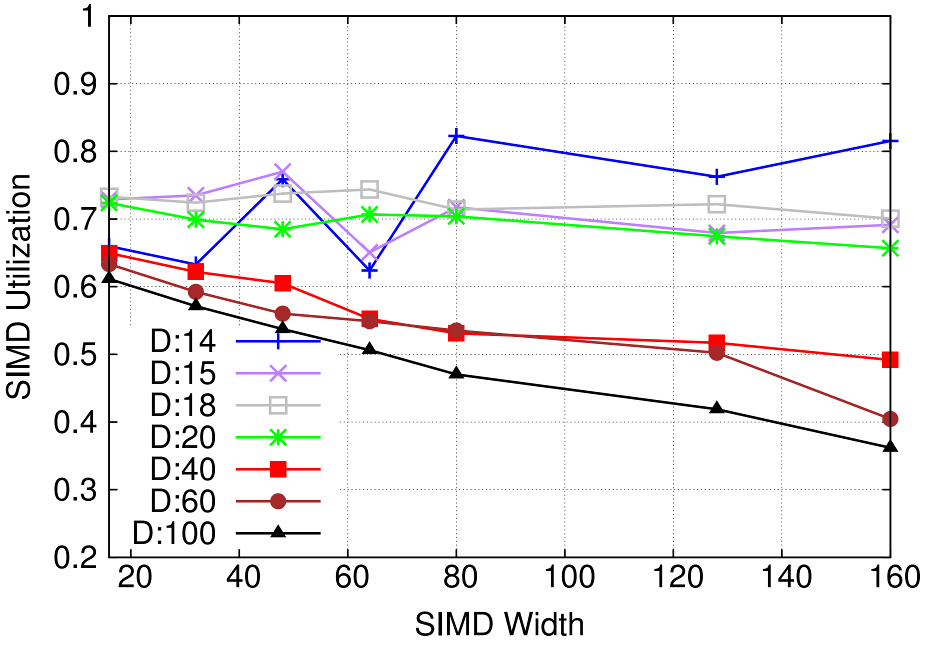


Fig. 4.12. SIMD width variation. As the SIMD vectors increase in size, they modify utilization both positively and negatively depending on the tree shape.

In Figure 4.12, we study the block size required to achieve good SIMD utilization across SIMD vector sizes. Specifically, we evaluate the SIMD utilization achieved for different SIMD widths when the ratio of block size of SIMD width is kept constant. Changing the SIMD width, for a given ratio of block size to SIMD width, improves vectorization till work generation saturates, at which point there are not enough nodes to exploit the large amount of slots made available by the larger SIMD vector widths and data blocks. We observe that SIMD utilization remains high for fuller trees with maximum depth of 14, 15, and 18. As trees get deeper, the inherent sparsity in the trees causes gradual degradation in SIMD utilization achieved.

4.5 Conclusion

With the re-expansion algorithm in place, the graph proves that good utilization is obtained for the trees generated. These results, along with results from the previous section guarantee a low bound on the utilization benefits obtained from vectorization.

5. SUMMARY

In this thesis, we have shown that we can successfully transform a class of recursive functions using several scheduler techniques.

In the first chapter, we transformed a serial tree-traversal algorithm into a vectorized tree traversal algorithm using point-blocked code. This enables us to parallelize execution of such serial trees using vector-processors like SIMD which are widely available on most architecture platforms. This stage is called *Work Generation* and does not guarantee and utilization benefits by itself. Hence, we next explore how to improve the utilization of SIMD units for recursive trees by introducing new scheduling techniques to improve utilization, both by reorganizing existing parallel computations In the first technique *Single Step Re-Expansion*, we re-expand the SIMD stream by searching for more work from around the current node's parents. The second approach- *Re-expansion*, is something like a "mode swapping" BFS, where you have some set of points in our block, and we can either execute the points in that block in depth-first manner like how it is done in point blocking or in breadth-first manner, swapping back and forth to maintain the block's population. These techniques work well for most recursive algorithms and are effective in maintain a good utilization ratio for most groups of trees.

In cases where both the above policies are not effective, we adopt restart mechanisms to improve utilization. In some recursive functions, when there is not much work available down the tree, multiple re-expansion calls need to be triggered relative to the depth of the tree. In such cases, it is more efficient to look at the upper part of the tree to generate work. So, in Restart technique, we start operating on another un-processed block of nodes from the *work generation* block using the techniques explained above and merge the two blocks at some point to form a single new block. This technique is very effective for extremely stringy trees. Collectively, these three scheduling policies can be used with various recursive applications to vectorize them and maintain good utilization ratios.

In the next chapter, we provide a theoretical limit for SIMD utilization improvement that we get with our scheduling techniques. To support this claim, we explore the various possible tree traversal shapes that out scheduler can vectorize and profiles its performance for the same We use a uniform distribution curve to generate random tree topologies and show that our scheduler maintains utilization rations greater than 50% for trees with depths up to 10 times the full-tree depth, for a given number of nodes. Vectorization results obtained over 1000 trials for various tree configurations guarantee a low bound on the utilization benefits obtained with our schedulers.

6. RELATED WORK

Several programming languages for multicore systems explore the use of task parallel constructs that correspond to the execution of computation trees. This includes Cilk [12–14], Thread Building Blocks [15], Task Parallel Library [16], OpenMP [17], and X10 [18]. Typical vectorization considerations in these models has focused on *inner vectorization* where the base case is vectorized. In these cases, the induction case runs as context-independent threads on one or multiple processors and the base case is executed as a vector unit. Our work focuses on an analysis of the vectorization opportunities in the recursion itself. One related SIMD optimization technique, Raja is focused on enabling SIMD optimization of loop programs prevalent in scientific applications in a performance portable fashion.

For task parallel programs running on multicore machines, many variants of work-sharing and work-stealing schedulers provide theoretical guarantees of high utilization [19–23]. In work-sharing schedulers, all the tasks that are ready to execute are put in a shared pool and all workers take work from this pool when they need work. In work-stealing schedulers, the pool is distributed among a worker and workers "steal" work from other workers' pool when it runs out of its own work. For work stealing with multiple cores in the case of CILK, each processor has a stack that stores suspended frames of the current processes running on the cores. But these stacks are like dequeues and the suspended state can be removed form both ends. While each processor can remove frames from the stack from the same end that it inserts them in, other free processors can "steal" from the other end and begin execution. This keeps all the processors busy.

In both work-sharing and work-stealing, the goal is to keep all workers busy — however, since these strategies are designed for multicores, workers do not have to work in a SIMD fashion. To the best of our knowledge, our work is the first to characterize theoretical performance of schedulers that run task parallel programs using SIMD machines.

The *work-first strategy* in Cilk [13] is similar to our depth-first execution scheme: a processor encountered a task immediately begin recursive execution of the task. The *help-first strategy* developed by Guo [24] is similar to out breadth-first re-expansion strategy. They propose combining work-first and help-first strategies to combine faster work dissemination as compared to a strict work-first model. This work does not involve a systematic analysis of benefits such a combined approach for arbitrary programs. More importantly, this work focused on optimizing task parallelism in the context of independent threads that constitute a multicore system. We are not aware of any prior work on systematic analysis of data parallelism and associated vectorization potential in recursive programs.

REFERENCES

REFERENCES

[1] NVIDIA, *CUDA*, 2015. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[2] J. Flesch, F. Thuijsman, and O. J. Vrieze, "Recursive repeated games with absorbing states," *Mathematics of Operations Research*, vol. 21, no. 4, pp. 1016–1022, 1996. [Online]. Available: http://dx.doi.org/10.1287/moor.21.4.1016

[3] K. Rocki and R. Suda, "Parallel minimax tree searching on gpu," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 449–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882846

[4] D. Saougkos, A. Mastoras, and G. Manis, "Fine grained parallelism in recursive function calls," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II*, ser. PPAM'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1007/9783642315008_13

[5] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," *SIGPLAN Not.*, vol. 34, no. 8, pp. 72–83, May 1999. [Online]. Available: http://doi.acm.org/10.1145/329366.301111

[6] NVIDIA, "Cuda," 2015. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/ files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

[7] *Cilk 5.4.6 Reference Manual*, Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, Nov. 2001. [Online]. Available: http://supertech.lcs.mit.edu/cilk/manual-5.4.6.pdf

[8] C. E. Leiserson and A. Plaat, "Programming parallel applications in Cilk," *SIAM News*, vol. 31, no. 4, pp. 6–7, May 1998.

[9] *MinMax Theorem*, March 2015. [Online]. Available: http://www.princeton.edu/ achaney/tmve/wiki100k/docs/Minimax.html

[10] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503223

[11] B. Ren, T. Poutanen, T. Mytkowicz, W. Schulte, G. Agrawal, and J. R. Larus, "Simd parallelization of applications that traverse irregular data structures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494989

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, May 1998. [Online]. Available: http://doi.acm.org/10.1145/277652.277725

[14] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with exceptions in jcilk," *Science of Computer Programming (SCP)*, vol. 63, no. 2, pp. 147–171, Dec. 2006.

[15] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[16] *The Task Parallel Library*, Oct 2007. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[17] *OpenMP Architecture Review Board*, May 2008. [Online]. Available: http://openmp.org/wp/

[18] *The X10 Programming Language*, Mar 2006. [Online]. Available: www.research.ibm.com/x10/

[19] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM JOURNAL ON APPLIED MATHEMATICS*, vol. 17, no. 2, pp. 416–429, 1969.

[20] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, pp. 201–206, 1974.

[21] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[22] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129. [Online]. Available: http://doi.acm.org/10.1145/277651.277678

[23] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads." in *SPAA*, P. B. Gibbons and M. Adler, Eds. ACM, 2004, pp. 235–244. [Online]. Available: http://dblp.uni-trier.de/db/conf/spaa/spaa2004.html

[24] Y. Guo, R. Barik, R. Raman, and V. Sarkar, “Work-first and help-first scheduling policies for async-finish task parallelism,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161079