ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **15**, Number 3, 2012, 215–228

A SIMD Approach to Thread Matching for Interleaved Multithreading

Lucian PETRICĂ

IMT Bucharest
University "*Politehnica*" Bucharest
Faculty of Electronics, Telecommunication
and Information Technology, DCAE Department
E-mail: `lucian.petrica@dcae.pub.ro`

Abstract. Interleaved multithreading processors offer improved performance and power efficiency in a multithreading environment compared to standard CPUs by allowing multiple threads to share a single processing pipeline. However, resource contention is a natural result of such a system and can determine how well the overall thread group performs on the processor. Selecting threads which perform well together has proven to be a difficult problem to solve because it is computationally intensive and grows rapidly in complexity with the number of threads in a system. We propose a vector algorithm for this task which is able to significantly outperform a scalar implementation and which shows improved scaling characteristics.

1. Introduction

Interleaved multithreading is an already proven technology, having been implemented in Sun's Niagara line of server processors for several years. Independent studies have shown the interleaved approach to provide better performance at lower power consumption for server applications, yielding significant energy savings overall compared to traditional server processors. The energy saving in particular has made the interleaved processing paradigm a topic of research and development in the embedded space [1], with Software Defined Radio as a potential application.

One significant drawback of the interleaved approach is the increased sensitivity to thread characteristics. Whereas in a single-threaded processor thrashing of a cache or

saturation of a multiplier affects only the running thread, in an interleaved processor one thread can slow down other threads which run alongside it. Furthermore, threads which behave well on their own may work badly together because they contend for resources [5, 4]. Intuitively, a solution to this problem is to measure thread interaction and group threads together based on the performance of the whole group. Several papers have touched on this subject and the problem has been shown to be extremely difficult, especially in server applications which work with tens to hundreds of threads [6, 7, 8]. A heuristic work-around has shown good potential by estimating group performance from single-threaded performance of each thread in the group [5].

We propose to revisit the problem of thread matching in the context of embedded applications and parallel processing. Embedded applications are generally much less threaded than server applications but performance requirements are just as strict. Additionally, embedded applications for mobile devices must work within very tight energy budgets. This has led to the adoption of interleaved multithreading in some niche markets of embedded processing. The target platform is a wide-SIMD vector processor. The proposed vector algorithm for thread matching is able to utilize the platform SIMD resources to do the partitioning of threads into groups which can be scheduled sequentially on a single processor or simultaneously on several interleaved processors within the system.

The contribution of this paper is the first vector algorithm for solving the thread matching problem. The paper also makes detailed analysis of the complexity of this algorithm with regards to its input parameters, and suggests ways in which vector processor hardware can be improved in order to provide better performance for this particular task. The rest of the paper is structured as follows. Section 2 provides an overview of the thread matching problem and defines the algorithm for solving it. Section 3 describes the target platform and states size requirements of the algorithm, then gives the actual implementation of the algorithm for the target platform. Performance and profiling, as well as hardware optimizations, are presented in Section 4. Concluding remarks and avenues for future work are presented in Section 5.

2. Thread Matching

2.1. The Concept

A software system will contain a number of application threads, which we can think of as function calls, for simplicity. Each time the user causes a function to be called, a new thread is created and added to the pool of existing threads. Sometimes a particular function will have multiple instances running at the same time. This can occur for a multitude of reasons, for example if two system users are encoding video streams, there will be two simultaneous instances of the video codec. We can think of threads which are instances of the same function as having the same *thread type*. All threads with the same type have similar characteristics with respect to resource usage, computational intensity, cache behaviour and other indicators. Once we have measured these indicators for one thread of a given type, we can expect all

other threads of the same type to behave in a similar fashion. We can measure the computational efficiency of a thread type by measuring clocks per instruction while the thread is running on the processor.

An interleaved processor will execute multiple threads simultaneously, a *thread group*, sharing CPU cycles and processor resources among the threads. At each cycle, a thread is selected for execution and an instruction from that thread is inserted into the pipeline. Thus, clocks per instruction is no longer a characteristic of a single thread, but is a feature of the thread group. While we can measure individual progress of each thread in the group, like megabytes encoded or triangles processed, we can only measure computational efficiency for the group as a whole. Because all thread of the same type have the same characteristics, we can introduce the notion of *type groups*, which are groups formed of threads of specific types. We can expect all instances of the same type group to perform similarly with respect to performance and efficiency.

Thread matching is the problem of analyzing a pool of runnable threads and arranging them into type groups such that the overall predicted efficiency is maximized. For better illustration of the problem at hand, we will exemplify thread matching using a software defined radio processing pipeline for the WiMAX standard. The WiMAX transmitter is a multi-stage pipeline which takes a data sequence as input and produces an encoded and modulated symbol stream which is ready for transmission over the physical link. The pipeline includes an encoder, a mapper, an assembler and an inverse FFT. The encoder is itself a pipelined operation and includes a Reed-Solomon encoder, a convolutional encoder and an interleaver [3]. Overall, the processing chain contains 11 stages, each of which does a different type of computation and is thus a separate thread type. If we are transmitting data over several radio links at the same time, there will be multiple instances of the WiMAX transmitter and so multiple instances of each of the processing stages. Considering an example of 10 simultaneous links, we might have as much as 110 runnable threads, and each thread belongs to one of 11 thread types. The scheduler will then group these threads based on their types and then commit the groups to the available processors.

2.2. Problem Statement

Given a pool of N threads, a K-interleaved processing system and an objective function F, thread matching requires the partitioning of the thread pool into groups of K threads such that F is minimized. The collection of groups which result from the partitioning is called the schedule. In our particular case, F is the aggregate Clocks Per Instruction of the schedule, obtained by averaging CPIs of all groups within a schedule. Obviously this is a very difficult optimization problem and the number of possible schedules P grows rapidly with N in particular.

$$F = \frac{K}{N} \sum CPI(g_i) \tag{1}$$

$$i \leqslant \frac{N}{K} \tag{2}$$

$$P = \prod C_{N-jK}^{K} \tag{3}$$

$$j \leqslant \frac{N}{K} - 1 \tag{4}$$

This is the reason why thread matching is unfeasible in a server environment, where tens of threads, all of different types, can be available for scheduling at the same time. Even in an embedded environment with fewer threads, the problem is still a difficult one to solve generally. A more approachable problem is that of partitioning the pool itself into windows of 2K threads each, to be scheduled separately. This means we must solve for a pool of 2K threads, and the schedule consists of only two thread groups, g_1 and g_2, which are complementary. This scheme greatly reduces the computational complexity of the algorithm.

$$F = \frac{CPI(g_1) + CPI(g_2)}{2} \tag{5}$$

$$P = \frac{C_N^K}{2} = \frac{C_{2K}^K}{2} = \frac{(2K)!}{4K!} \tag{6}$$

Obviously, windowing the pool of threads in this way will, in most situations, yield a worse schedule than could be obtained by partitioning the whole pool, but the problem is now much less computationally intense and becomes approachable. Determining the exact loss of efficiency caused by the windowing is outside the scope of this paper.

2.3. The Objective Function

The objective of task matching is to increase the performance of the processor. We will choose Clocks per Instruction (CPI) as the metric for processor performance, where lower CPI is better. The algorithm assumes that the system can accommodate up to T distinct thread types and that the performance of all type groups has been already measured from hardware instruction and clock counters within the processor, at the first run of each particular group. It is of special interest what is the exact number of possible groups formed from the available T thread types, given that within a type group there may be multiple threads of the same type. This problem is an extension of the concept of combinations, and it can be easily proven that the number, denoted G, is given by [7].

$$G_T^K = \frac{(T+K-1)!}{(T-1)!K!} \tag{7}$$

$$G_T^K = C_{T+K-1}^K \tag{8}$$

The particular representation of each CPI measurement is irrelevant to this algorithm as long as it is achieved through a monotonically increasing transformation from the space of actual CPI values. Strict monotonicity is not a requirement but will prevent CPI aliasing and lead to better overall scheduling performance.

Algorithm 1 Generic Thread Matching

```
for each pair of combinations of K threads

    retrieve thread types
    compute type group indexes
    retrieve CPI

compute minimum CPI
return combination
```

2.4. The Algorithm

The first step is to define algorithm parameters, inputs and outputs, as presented in Table 1. In addition to parameters K and T which have already been defined, the algorithm takes as input two arrays. ThreadTypes is an array containing the thread types of the threads within the runqueue, such that the first element in ThreadTypes is the type of the first thread in the runqueue and so on. CPITable is an array of CPI values corresponding to all possible type groups. The algorithm outputs an array representing the resulting schedule by depth indicators in the runqueue.

Table 1. Algorithm Inputs and Outputs

Name	Direction	Type	Size
K	Input	Integer	1
T	Input	Integer	1
ThreadTypes	Input	Integer Array	2K
CPITable	Input	Integer/Float Array	G_T^K
Schedule	Output	Integer Array	2K

The next step is to express the algorithm in its most generic form. Algorithm 1 is a pseudocode description of the algorithm. At each step we analyze a different possible schedule, which is a permutation of the threads in the window. It must be noted that all permutations of elements within a thread group are equivalent, because they will be run together on the same processor. Because of this property, we are actually looking for two complementary combinations of K threads.

Since we are interested in the types of threads, we read these from the runqueue. In the next step we compute the indexes of the two resulting thread groups in the CPI table and we retrieve the CPI. Finally, we find which schedule performs better than all the others and store it. The first thing to notice is that finding the CPI for each schedule is independent of all others. Secondly, computing the minimum CPI is a reduction operation and requires global data interaction, and is therefore not parallel, but it can be done efficiently in logarithmic time.

This analysis permits us to redesign the algorithm to utilize the available parallelism. We will assign each CPI computation to a worker process. We look to equation (6) to give us the maximum number of workers which the algorithm can utilize. Each will require a copy of algorithm inputs. Our ability to extract parallelism

Algorithm 2 SIMD Thread Matching

```
send inputs to all workers
each worker must
    retrieve thread types
    compute type group indexes
    retrieve CPI
compute minimum CPI
return combination
```

from the second step depends on the target platform, so this issue will be analyzed further in Section 3. An outline of the parallel algorithm is presented in Algorithm 2.

3. Implementation

3.1. Target Platform

The SIMD algorithm will be implemented on the BEAM-Connex procesing system. This platform has been described in more detail previously in [2, 1]. It consists of a 4-interleaved multithreading processor coupled to a 128-cell SIMD engine. Each cell has access to a 1024-word local memory, and all cells execute the same instruction at the same time. Cells can be enabled or disabled individually by software to allow for conditional execution. The processor can distribute scalar data to all the cells simultaneously, and specific reduction operations can be performed on whole arrays of data within the SIMD engine, such as logical OR and addition, through the use of a logarithmic tree of arithmetic logical units. Inter-cell communication is only side-to-side. The SIMD cells operate on 16-bit integer values, which is also the width of the local memory. A DMA engine serves the SIMD engine and can load data from main memory into the local memories.

3.2. Mapping The Algorithm

The algorithm can be implemented directly on the target platform by having each SIMD cell map to a different worker process. The limited SIMD processor size will only allow direct mapping of the algorithm up to certain values of K and T. We can see in Table 2 the maximum values of algorithm parameters which allow direct mapping to the target platform. Memory requirements are computed for the maximum allowable value of T and include auxiliary data.

Table 2. Allowed Parameters

K	2	3	4	5
Cells	3	10	35	126
T_{max}	44	17	10	8
Memory	1007	993	746	830

As we can see, direct mapping works for K up to 5. For larger values of K, the we require more cells than are physically available, and the algorithm needs to be broken up into consecutive chunks. The maximum allowable value of T is given by the value of K and the size of the cell local memory, as expressed by equation (7). Accommodating larger values of T is more difficult than simply paging the local memory. Because each cell computes its own indexes in the CPI table and this index depends on the particular configuration of threads in the runqueue, different cells could point to different pages. Additionally, the two indexes computed by each cell could themselves point to different memory pages. Resolving such a situation brings a larger performance penalty than is acceptable for the task at hand and as such we choose to disregard this possibility and treat T_{max} as a hard constraint.

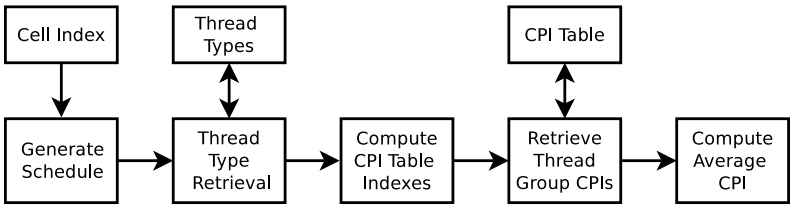

Fig. 1. Worker Mapped to a SIMD Cell.

3.3. Distribution of Inputs

The first step of the SIMD algorithm for thread matching is making a copy of algorithm inputs in each cell. There are two ways to transfer data from main memory into the local memory of the SIMD engine. The first method if through the distribution network, which allows the scalar processor to copy the contents of one of its registers into all the SIMD cells simultaneously. This way of transferring data naturally fits this particular use case, but the data must pass through the controller data cache and with as much as two kilobytes of data transferred in this way, it can cause thrashing and reduce overall system performance.

The second method is programmed DMA transfers. The DMA will transfer a region of main memory into the SIMD cells such that the first cell receives the first two bytes of the designated main memory area, the second cell receives the next two bytes and so on. It is not possible to transfer the same data into all the cells of the SIMD engine unless the data is duplicated 128 times in memory and then a DMA transfer is initiated. Generally, the main advantage of DMA transfers is that they can occur in the background while other computation takes place, and are thus faster than distribution network transfers. In this particular use-case there are several drawbacks to this approach. First, data needs to be duplicated 128 times in memory. This is a particular problem for the CPI table, which can grow to nearly 2 kilobytes. If we are to duplicate this data we would waste nearly 256 kilobytes of main memory. Secondly, this large amount of data would need to be transferred from main memory into the SIMD engine every time the algorithm is executed. While the transfer time could

be masked by other computations, of greater concern is the energy consumption of transferring such large amounts of data over a DDR bus and across chip pins.

Table 3. Transfer Rates

Mode	Native Rate	Redundancy	Actual Rate
DMA	16 B/cycle	128	0.125 B/cycle
Distribution	2 B/cycle	1	0.070 B/cycle

In Table 1 we can see there are two integer array inputs to the algorithm, which have to be copied to the SIMD engine. The total amount of data to be copied as well as transfer rates for the two transfer modes are listed in Table 3. Transfer rates over DMA are significantly faster than over the distribution network, and it is nonblocking which means it can overlap with other computation. The actual transfer rate of the distribution network is lower than the native rate because it needs to be fed from the data cache and also the transfer is executed within a loop, which adds some overhead for index calculation and the jump itself. The optimal solution for this step of the algorithm will most likely be determined by system considerations such as amount of memory and the energy budget.

3.4. Generation of the Schedule

The algorithm proposes that each cell will work on different complementary combinations of threads, which together form a schedule. We must either generate all the combinations in the controller and then load each cell with its corresponding schedule, or generate the corresponding schedule within the cell itself, using the cell's index number as the combination's lexicographical index. In essence, we must convert a number to the combinatorial number system. A theoretical and algorithmic overview of combinatorial number systems is given in [9]. To better illustrate the implementation of this stage of the algorithm, we will follow an example for K=2. All cells execute the same program at this stage so this algorithm is repeated everywhere.

We can see in Table 3 the correspondence between combinations and their lexicographical index. The last columns list "reverse" combinations, which are obtained according to equation (9). We can also observe the properties of the reverse combination sequence. Any combination element can take values smaller or equal to the element to the left of it. From this we derive the relationship between RIndex and RComb, which is given by equation (10).

$$\begin{gathered} 0 \leq i < K \\ 0 \leq Comb[i] < 2K \\ RComb[i] = 2K - 1 - i - Comb[i] \\ RIndex = C_{2K}^{K} - Index - 1 \end{gathered} \tag{9}$$

$$\begin{gathered} RIndex = RComb[0] + \sum_j j \\ j \leq RComb[1] \end{gathered} \tag{10}$$

Table 4. Combinations for K=2

Index	Comb	RComb	RIndex
0	(0,1)	(2,2)	5
1	(0,2)	(2,1)	4
2	(0,3)	(2,0)	3
3	(1,2)	(1,1)	2
4	(1,3)	(1,0)	1
5	(2,3)	(0,0)	0

Table 5. Accumulators for K=2

Level	acc[1]	acc[0]
0	0	0
1	1	1
2	3	2

The accumulators in Table 5 represent the terms in equation (10) for K=2. The "level" corresponds to the value of RComb[0] and RComb[1]. We start from the top Level and we iterate, comparing RIndex to acc[1]. If RIndex is greater or equal, then RComb[1] equals the current level. We subtract acc[1] from RIndex, and we repeat the process, comparing with acc[0]. We do this for all levels. In a scalar implementation we would stop once we found all elements of RComb, but since this program is SIMD in nature, program flow must be static so we execute for all accumulators and all levels even though computation may be unnecessary in some cells. Once we found RComb we can easily obtain Comb. Time complexity of schedule generation is $O(K^2)$ and space complexity is $O(K)$.

3.5. Retrieving Thread Types

We will now use generated complementary combinations to partition the window into two type groups. The process is exemplified below and involves using each element in the combination as an offset into the ThreadTypes input array and reading the type of algorithm at that address. Implementation of this is simple, involving $O(K)$ operations, most of them additions, to form addresses from the base and offsets, and local memory reads, to extract the type groups from the window. Space complexity of thread type retrieval is $O(K)$.

$$\begin{aligned} (1356) &\rightarrow (2\mathbf{2}3\mathbf{4}5\mathbf{5}\mathbf{6}\mathbf{7}) \rightarrow (2456) \\ (0247) &\rightarrow (\mathbf{2}2\mathbf{3}\mathbf{4}\mathbf{5}56\mathbf{7}) \rightarrow (2357) \end{aligned}$$

3.6. Computing CPI Indexes

This operation is the inverse of generating a combination from its lexicographical index. At first the problem would seem different because there can be duplicate elements in the type group, unlike a lexicographical combination, however it can be easily shown that a type group of K elements from T types exhibits the same properties as a combination of K elements from a group of $T + K - 1$. Thus, we can handle the index computation using the same methods shown before in equation (10), when generating the schedule. Index computation has a time complexity of $O(TK)$ and the same complexity $O(K)$. Note that we must compute an index for each of the two type groups formed from the initial schedule.

3.7. Finding the Minimum CPI

If we have the indexes for the two type groups of each cell, we can read the CPI values of each type group and average them, which gives us the CPI of the schedule as a whole. At this point, we have obtained the CPIs of all schedules which can be formed from the thread window, with each cell holding the CPI value of a different schedule. Now we must find out which schedule performs the best, and thus has the lowest CPI value. To do this, we will use the SIMD engine's inter-cell communication network to shift all CPI values into the first cell, which keeps track of the lowest CPI value encountered and the index of the cell which contained it. After all the CPI values have been shifted into the first cell, we know which cell has the minimum CPI, and consequently we know which schedule is optimal for the given thread window.

4. Performance

4.1. Initial Evaluation

Performance was measured for all values of K which we were able to fit in the target vector processor. Our benchmark was a scalar implementation of the same algorithm, running on a single thread on the controller. Two versions of the vector algorithm were measured, one with DMA input copying and the other utilizing the distribution network. All runs had T set to 8 which was the largest value supported for K=5. The measurements were carried out on a cycle-accurate simulator of the vector processor generated from Verilog code by Verilator. The compiler used to generate code for the algorithm was GCC 4.4.5, which was ported to the target architecture. Because of difficulties in porting GCC to a complicated heterogeneous architecture such as the one used here, it was unable to perform proper optimization on vector code, which is compiled with -O0, while scalar code is compiled with -O3.

Benchmarking results are shown in Fig. 2. As would be expected, for K=2 there is no speedup from the vector algorithm because parallelism is low with just 3 cells used, and initialization of the vector processor, which includes input copying, takes too long. For larger values of K, we begin to see speedup when compared to the performance of the scalar implementation. Another point to be made is the significant performance advantage of the DMA implementation over the one which uses the distribution network. This is because for the DMA implementation copying the inputs into the SIMD engine completely overlaps with schedule generation.

4.2. Optimization

Profiling results are shown in Fig. 3 for all acceptable values of K and various values of T. We can see that input copying takes up a significant percentage of execution time, and is dominant for larger values of T. Computation of the minimum CPI value is dominant for smaller T and larger values of K. Further optimization will focus on these functions.

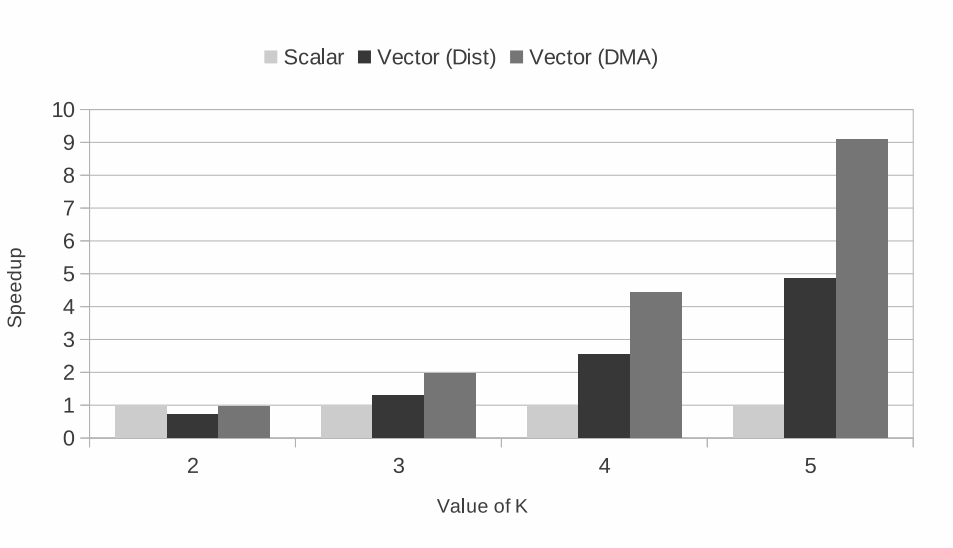

Fig. 2. Algorithm Performance.

Distribution DMA We have seen that a significant part of the total runtime of the vector algorithm is spent copying the input arrays into the SIMD engine through the distribution network. One potential solution is to use the DMA which is able to hide the transfer behind other computation and performs significantly better. However, the DMA solution, as it is implemented, requires duplication of the input arrays 128 times which will in most cases be unacceptable to the system designer because of memory and power constraints. It is necessary to find a solution to utilize the distribution network in a similar fashion to the DMA engine. We have implemented a modification to the IO system of the SIMD engine which permits the DMA, in addition to the controller, to serialize data into the distribution network directly when so instructed. Hardware overhead of this solution is minimal, and it solves several problems. First, transfers through the distribution network no longer occupy CPU time. Second, because transfers are no longer driven by software, the full throughput of the distribution network can be utilized. Third, the input arrays no longer pollute the controller data cache when being transferred into the SIMD engine.

We must make a note of an intermediate solution which requires even less hardware modification. We can reduce memory requirements by having the DMA engine duplicate the data as it arrives from the memory controller in a way that will produce the desired local memory contents in the Connex array, but in transposed form. We then do a matrix transposition to rearrange data inside the local memories. This solution is conceptually inferior to the Distribution DMA because it requires code to be executed for the transposition and therefore cannot be non-blocking.

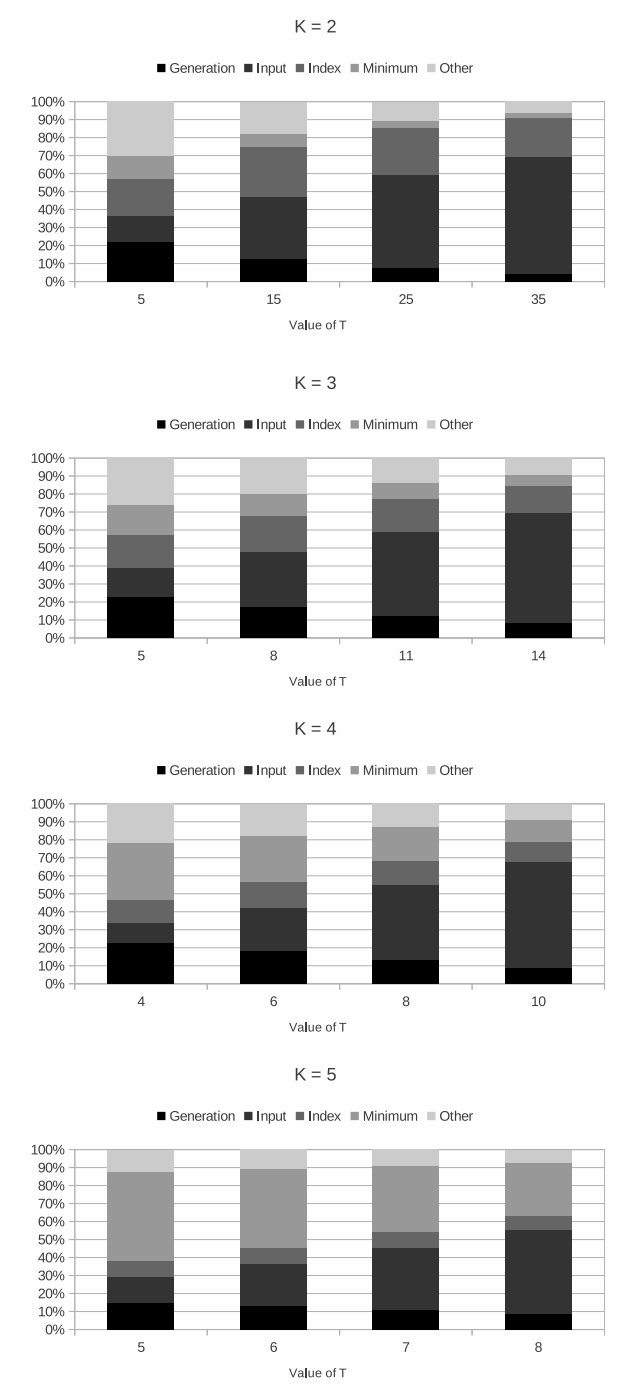

Fig. 3. Profiling Results.

Reduction Network Min/Max Another big contribution to algorithm runtime is the computation of the minimum CPI, which is done serially in the SIMD array and benefits from no parallelism. The SIMD array does however have a reduction network which allows it to compute sums of data from all cells at once. It would be beneficial if we could add the function of computing the minimum of the values contained in all cells, which would allow fast computation of this particular step in the algorithm. Implementing this change adds about 30% overhead in area for the reduction network, and overall less than 5% of total area overhead.

Performance Reevaluation We have repeated the previous tests in order to evaluate the impact of the changes. The reduction network optimization yields as much as 50% increase in speedup. Adding to this the distribution network optimization takes the overall speedup for K=5 to over 20, which is over four times the initial speedup of the vector algorithm over the scalar one. The improved results are presented in Fig. 4.

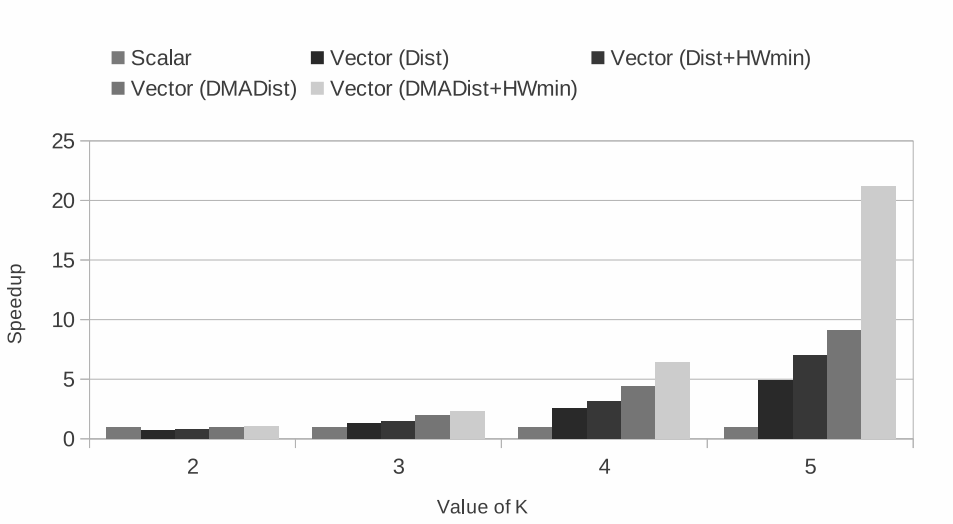

Fig. 4. Optimized Performance.

5. Conclusion and Future Work

We have implemented a parallel algorithm for finding the best scheduling of threads for execution on an interleaved multithreading processor. Computation of the algorithm is done by multiple cells of the SIMD engine in the BEAM-Connex target platform. An initial performance evaluation was used to identify the speedup of the SIMD implementation over the scalar implementation of the algorithm, for various algorithm parameters. The initial evaluation included profiling data which

allowed us to identify key areas where improvements could be made to the target platform. These improvements were implemented and subsequent evaluations show significant gains in speed over the initial results.

Future work will have to include a method for handling input parameters of larger sizes than the ones discussed in this paper. Larger values of K require more cells than are physically available and thus a method to fold the algorithm is necessary. Larger values of T are more problematic but one solution is to represent CPI values with less precision. Currently a CPI table entry occupies a whole 16-bit word in the cell local memory, but if we were to reduce CPI precision to 4 bits we could hold CPI tables four times larger. The effect of this loss of precision on the system-wide performance of the scheduler making use of our algorithm is an interesting topic for further research.

References

[1] Codreanu V., Hobincu R., *Performance gain from data and control dependency elimination in embedded processors*, ISETC, 2010.

[2] Codreanu V., Petrică L., Hobincu R., *Increasing vector processor pipeline efficiency with a thread-interleaved controller*, ICSTCC, 2011.

[3] Roca A., *Implementation of a WiMAX simulator in Simulink*, PhD Thesis, University of Vienna, 2007.

[4] Fedorova A., Seltzer M., Small C., Nussbaum D., *Throughput-oriented scheduling on chip multithreading systems*, Technical Report, Harvard University, 2004.

[5] Fedorova A., Seltzer M., Small C., Nussbaum D., *Chip multithreading systems need a new operating system scheduler*, *Proceedings of the 11th workshop on ACM SIGOPS*, 2004.

[6] Snavely A., Tullsen D., *Symbiotic Jobscheduling for a Simultaneous Multithreading Machine*, ASPLOS IX, November 2000.

[7] Snavely A., Tullsen D., Voelker G., *Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor*, SIGMETRICS, 2002.

[8] Parekh S., Eggers S., Levy H., Lo J., *Thread-sensitive Scheduling for SMT Processors*, `http://www.cs.washington.edu/research/smt/`, 2000.

[9] Knuth D., *The Art of Computer Programming*, Volume **4**, Fascicle 3: *Generating All Combinations and Partitions*, 2005.