2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors

Scalar Processing Overhead on SIMD-Only Architectures

Arnaldo Azevedo, Ben Juurlink
Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, The Netherlands
Email: {A.P.PereiradeAzevedoFilho, B.H.H.Juurlink}@tudelft.nl

***Abstract*—The Cell processor consists of a general-purpose core and eight cores with a complete SIMD instruction set. Although originally designed for multimedia and gaming, it is currently being used for a much broader range of applications. In this paper we evaluate if the Cell SPEs could benefit significantly from a scalar processing unit using two methodologies. In the first methodology the scalar processing overhead is eliminated by replacing all scalar data types by the quadword data type. This methodology is feasible only for relatively small kernels. In the second methodology SPE performance is compared to the performance of a similarly configured PPU, which supports scalar operations. Experimental results show that the scalar processing overhead ranges from 19% to 57% for small kernels and from 12% to 39% for large kernels. Solutions to eliminate this overhead are also discussed.**

***Keywords*-Computer architecture; Datapath; SIMD processing; SIMD overhead;**

I. Introduction

The industry has moved towards multicore architectures as increasing instruction-level parallelism (ILP) has yield diminishing returns. Multicore processors are an efficient way to increase performance while avoiding the power wall, since less power is wasted on extracting and exploiting ILP. The efficient use of multicores, however, relies on the presence of explicit thread-level parallelism (TLP) in applications.

Multimedia as well as other applications typically exhibit significant amounts of data-level parallelism (DLP). DLP can be exploited in a power-efficient manner by means of Single-Instruction Multiple-Data (SIMD) operations. SIMD units have been used by high-end processors to accelerate multimedia applications. The Sony-Toshiba-IBM (STI) Cell processor brought this concept further and introduced a multi-core processor with 8 SIMD-only cores. To perform a scalar operation on the Cell Synergistic Processing Elements (SPEs), the scalar operands have to be shifted to the so-called preferred slot and the scalar result has to be merged with the rest of the vector.

Although originally designed for multimedia and gaming, the Cell processor has also been used as a basic block of high-performance and supercomputers. For example, it is part of the first supercomputer to run Linpack at a sustained speed in excess of 1 Pflop/s [1]. The computational performance and power efficiency of the Cell processor make it a suitable option to accelerate a wide range of applications.

Unfortunately, not every parallelizable application benefits significantly from SIMD processing. Sorting, information retrieval (e.g. histogram), and other kernels do not exhibit substantial amounts of DLP. These kernels, however, belong to important applications that cannot be neglected. Furthermore, even if a kernel is vectorizable using SIMD instructions, it often contains scalar operations.

In this paper we present an evaluation of the SPEs for parallelizable kernels/applications that do not benefit significantly from SIMD processing. The objective of this work is to identify the overhead introduced by the lack of scalar operations and the situations that cause this overhead. In other words, the goal is to evaluate if the SPE (or SIMD-only processors in general) would profit significantly from a scalar datapath. We study the impact of compiler-managed scalar and unaligned data for different applications. If significant overhead is incurred, it can justify modifications to the SPE architecture. The performance degradation for scalar computations can justify the area overhead of a scalar unit or other support for scalar operations.

The main contributions of this paper are:

- to the best of our knowledge, we are the first to quantify the overhead caused by the lack of hardware support for scalar operations on SIMD-only architectures such as the Cell SPE,
- we identify the sources of this overhead,
- we discuss possible solutions with low area overhead.

Previous work mainly focused on SIMD overhead, i.e., the overhead needed for bringing the data in a form amenable to SIMD processing. Ranganathan et al. [2] found that on average 41% of all instructions are SIMD overhead. Compiler techniques to reduce overheads related to data permutations, strided accesses, and alignment constraints were presented in [3], [4], and [5], respectively. Hardware techniques to reduce packing/unpacking and data rearrangement overhead were proposed in [6]. Alvarez [7] measured overhead of unaligned access for video processing and proposed splitting the memory bank into two to support unaligned data accesses. All these works, however, focused in SIMD overhead, not the overhead related to scalar processing on SIMD-only architectures.

1063-6862/09 $25.00 © 2009 IEEE
DOI 10.1109/ASAP.2009.12

This paper is organized as follows. Section II discusses the architecture of the Cell SPE, focusing on the characteristics that introduce scalar processing overhead. In Section III the methodologies applied to determine the scalar processing overhead are presented. It is followed, in Section IV, by a brief description of the evaluated kernels/applications. Experimental results are presented in Section V and conclusions are drawn in Section VI.

II. Cell processor architecture

The Cell Broadband Engine [8][9] is a heterogeneous multi-core processor designed for multimedia and game processing. It consists of one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPEs) connected by four 16B-wide data rings.

The PPE is a simplified version of the PowerPC processor family. It is based on IBM's 64-bit Power Architecture [10] with 128-bit vector media extensions. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications. The PPE is dual-threaded and has a two-way in-order execution pipeline unit with 23 stages. The PPE supports a conventional two-level cache hierarchy with 32KB L1 instruction and data caches and a 512KB unified L2 cache.

The SPEs are tailored for multimedia processing and are single-threaded, non-preemptive, two-way in-order processors. One issue slot can contain fixed- and floating-point operations and the other can contain loads/stores, and byte permutation operations, as well as branches. Branches are hinted by software and miss-predicted branches have a penalty of 18 cycles. The register file contains 128 128-bit wide registers. All instructions are SIMD and they operate on 128-bit vectors with varying element width, i.e. 2×64-bit, 4×32-bit, 8×16-bit, 16×8-bit, or 128×1-bit. Data should be 128-bit aligned and there is no hardware support for scalar operations.

SPEs can only access data and code stored in its 256KB Local Store (LS). The LS is mapped onto the main memory address space to allow LS-to-LS communication, but this memory (if cached) is not coherent in the system. Data and instructions are transferred, in packets of at most 16 KB, between the LS and main memory by explicit DMA commands, executed by the SPEs' DMA unit. Data communication can be performed in parallel with computation. A double buffering technique can be employed to hide the DMA transfer latency.

All DMA transfers and LS access are aligned to 128 bits. This design decision of not supporting scalar and unaligned operations was taken to reduce the control complexity and to eliminate several stages from the critical memory access path [9]. To manage scalar and unaligned data, the compiler inserts shuffle and shift instructions to align or allocate the data in the preferred slot. The preferred slot is the leftmost word in a 128-bit quadword. The SPE compiler manages scalar operations by first moving both operands to the preferred slot of a register. Next, the SIMD operation corresponding to the scalar operation is performed, and then the result is merged with the original content of the register. The shuffle moves the result to the destination slot and writes the data back to the register file. For instance, memory addresses for loads and stores, branch conditions and branch addresses for register-indirect branches are placed in the preferred slot.

Note that for operations that involve only scalar variables, the compiler can place these variables in memory locations congruent with the preferred slot, at the price of wasting memory. If, however, the scalar is an element of a vector, then reorganization overhead is required.

III. Methodology

In this section, the methodologies to evaluate the scalar processing overhead are described. Since implementing an architectural model would require too much effort for an initial evaluation, a Large-Data-Type methodology is presented to study the effects of the compiler-generated overhead for support scalar operations. This methodology, however, is feasible only for relatively small kernels. To evaluate larger kernels, a second methodology called SPE-vs-PPE is presented.

A. Large-Data-Type Methodology

The Large-Data-Type (LDT) methodology highlights the differences between the current SPE and an SPE with support for scalar and unaligned operations. No influence from the DMA transfers will be considered in the performance measurements. The main micro-architectural resources that are the focus of this methodology are the register file, its communication with the Local Store (LS), and the execution datapath. To allow this, the workload should fit in the SPE LS.

The main overhead when processing scalar data is shuffle/shift instructions. Without such instructions, the behavior of the architecture is equivalent to a processor with support for scalar operations. To simulate the behavior of a scalar unit, the kernels have been modified as follows. First, the processing data types are modified to 128-bit wide vectors. In order to avoid shuffles and shifts, the actual data is stored in the preferred slot. Comparisons are performed using SIMD intrinsics to make the code compilable.

The methodology is illustrated using the functions depicted in Figure 1. The function adds the vector A to the reversed vector B and saves the result in vector C. Figure 1 (left) depicts the regular function. The regular SPE implementation requires that every position of the vectors A and B are aligned by shuffling before processing them. This is performed by the $rotqby$ and $shufb$ instructions and causes considerable overhead. The compiler-generated assembly code for $C[i] = A[i] + B[size - 1 - i]$; is depicted

```
Regular_Implementation(int A[], int B[],
                       int C[], int size){
   for (i=0; i < size; i++)
     C[i] = A[i] + B[size - 1 - i];
}
```

```
LDT_Implementation(vector int A[],
vector int B[], vector int C[], int size){
    for (i=0; i < size; i++)
      C[i] = spu_add(A[i],B[size - 1 - i]);
 }
```

Figure 1. Example of compiler-managed scalar computation (left) and Large-Data-Type methodology (right)

```
// $8 : i
// $11: address of A
// $4 : address of B + size - 1 - i
// $5 : address of C

a       $2,$8,$11    //Adds i to A to calc
                      distance from pref slot
lqx     $3,$8,$11    //Load A[i]
lqd     $10,0($4)    //Load B[size - 1 - i]
lqx     $7,$8,$5     //Load C[i]
cwx     $9,$8,$5     //Generate control for
                      inserting C[i]
rotqby  $3,$3,$2     //Rotate A to pref slot
rotqby  $2,$10,$4    //Rotate B to pref slot
a       $3,$3,$2     //Add A and B
shufb   $7,$3,$7,$9  //Shuffles result into
                      final position of C
stqx    $7,$8,$5     //Store C
```

```
// $7 : i
// $8 : address of A
// $4 : address of B + size - 1 - i
// $5 : address of C


lqx     $2,$7,$8 // Load A[i]
lqd     $3,0($4) // Load B[size - 1 - i ]


a       $2,$2,$3 // Add A and B

stqx    $2,$7,$5 // Store result in C[i]
```

Figure 2. Generated assembly from scalar computation (left) and Large-Data-Type methodology (right)

in Figure 2 (left). The LDT methodology is depicted in Figure 1 (right). The spu_add intrinsic adds two aligned vectors. This way, the compiler generates only the core instructions without any shuffling, with the same behavior as an SPE with scalar support would have. The generated assembly for LDT methodology is depicted in Figure 2 (right).

This methodology, however, increases the data footprints of the kernels, as the scalars are now four times larger. This could increase the pressure on the register file (RF) and the amount of data transferred over the LS-to-RF communication channel. Increasing the register pressure could mean that more variables need to be spilled to memory, while this would not be needed in an SPE with scalar support. To analyze the effect of this, we will compare the number of registers used in the LDT-emulated versions of the kernels to the number of registers needed in the original kernels. The effects on the LS-to-RF channel will be verified using the number of load/store stalls.

B. SPE-vs-PPE

The focus of the SPE-vs-PPE methodology is to evaluate large kernels and the effects of the compiler-managed scalar operations on a large piece of application. While the LDT methodology focuses on specific overheads, this methodology focuses on the regular case.

To compare the SPE against a regular processor, a natural choice is the Cell PPE. The PPE runs at the same clock frequency, and it has the same simplified organization of the SPE. There are fundamental architectural differences between the PPE and the SPE, however. The PPE is designed for control, while the SPE is designed for multimedia and game processing. The PPE has a traditional memory hierarchy, while the SPE accesses data and code in its LS. However, because both are in-order, dual-issue processors running at the same frequency with a 23-stage pipeline that commits fixed point operations every two cycles, a comparison between them can be made. Comparing the performance of SPE with the PPE on large kernels reveals the scalar (in)efficiency of the processors. To concentrate the comparison on the computational part of the processor, the data set of the kernels are limited to 32KB, the size of the L1 data cache.

IV. KERNELS

This section describes the kernels used in this study. The selection criteria for the kernels are that they should present scalar, hard to vectorize sections, but be parallelizable at the same time. These criteria are to match the type of applications likely to be ported to the Cell processor. There are two sets of kernels, “Small Kernels” and “Large Kernels”. Small

```
saxpy(int x[], int y[], int scalar,
                            int size){
   for (i = 0; i < size; i++)
      y[i] = scalar*x[i] + y[i];
}
```

Figure 3. Pseudocode for SAXPY

```
  for (i = ORDER-1; i < SIZE+ORDER-1; i++){
    accum=0;
    for (j = 0; j < ORDER; j++)
      accum += coefficients[j] * input[i-j];
    output[i-(ORDER-1)] = accum;
  }
```

Figure 4. Pseudocode for FIR filter

Kernels are the kernels used to highlight a particular characteristic of the SPE that causes scalar processing overhead, using the LDT methodology. The Large Kernels are used to compare the performance of the SPE with that of the PPE, using the SPE-vs-PPE methodology.

A. Small Kernels

The small kernels highlight characteristics of the SPE that introduce overhead for scalar computation. They access unaligned data, process scattered data, and make use of indirect addressing. For each characteristic two kernels were selected. Pseudo-C codes are listed for each kernel to allow the reproducibility of the experiments.

1) Saxpy: Basic Linear Algebra Subprograms (BLAS) is a linear algebra application programming interface. It is used for scientific computation and part of the LINPACK benchmarks. BLAS level 1 provides functionalities of the form $y = \alpha \times x + y$, where x and y are vectors and α a scalar, called the SAXPY kernel. This kernel was chosen to highlight the overhead of shuffle/rotate instructions used to move the data to the preferred slot. Its pseudocode is listed in Figure 3. A SIMD implementation of this kernel is possible, but requires some programming effort, because the vectors may not be aligned in memory.

2) FIR: Finite Impulse Response (FIR) filters are implemented by a convolution of the signal with the coefficients. It requires unaligned accesses to both the coefficient and input values, which requires more shuffle/rotate instructions than the previous kernel. Its pseudocode is listed in Figure 4.

3) QuickSort: Sorting algorithms are an important class of algorithms that are part of many applications. Sorting requires many comparisons, and aligned data accesses are hard to guarantee. QuickSort is one of the best known sorting algorithms. It works by recursively choosing a pivot and separating the values that are larger and smaller than the pivot. QuickSort pseudocode is listed in Figure 5.

```
quickSort (int a[], int lo, int hi) {
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    do{
        while (a[i]<x) i++;
        while (a[j]>x) j--;
        if (i<=j) {
            h=a[i]; a[i]=a[j]; a[j]=h;
            i++; j--;
        }
    } while (i<=j);

    //  recursion
    if (lo<j) quickSort(a, lo, j);
    if (i<hi) quickSort(a, i, hi);
}
```

Figure 5. Pseudocode for Quick Sort

```
void Merge( int a[], int b[], int c[],
    int m, int n ){
  int i = 0, j = 0, k = 0;
  while (i < m && j < n){
    if (a[i] < b[j])c[k++] = a[i++];
    else           c[k++] = b[j++];
  }
  while (i < m)  c[k++] = a[i++];
  while (j < n)  c[k++] = b[j++];
}

void merge_sort( int key[], int n ){
  int *w;
    for(i = 1; i < n; i *= 2 ){
      for(j = 0; j < (n - i); j += 2 * i )
        Merge(key + j, key + j + i,
                                 w + j, i, i);
      for (j = 0; j < n; j++) key[j] = w[j];
    }
}
```

Figure 6. Pseudocode for Merge Sort

4) MergeSort: Conceptually, a merge sort works as follows: Divide the unsorted list into two sublists of about half the size. Sort each sublist recursively by re-applying merge sort. Merge the two sorted sublists into one sorted list. Figure 6 depicts the MergeSort pseudocode.

5) Image Histogram: An image histogram is a type of histogram which acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value. The horizontal axis of the graph represents the tonal variations, while the vertical axis represents the number of pixels in that particular tone. This

```
for (i=0; i < img_height; i++)
  for (j=0; j < img_width; j++)
    histogram[ img[i][j] ]++;
```

Figure 7. Pseudocode for Image Histogram

```
for(i=1; i < img_height-1; i++)
  for (j=1; j < img_width-1; j++)
  {
      GLCM[ img[i][j] ][ img[i-1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i-1][j]   ]++;
      GLCM[ img[i][j] ][ img[i-1][j+1] ]++;

      GLCM[ img[i][j] ][ img[i][j-1]   ]++;
      GLCM[ img[i][j] ][ img[i][j+1]   ]++;

      GLCM[ img[i][j] ][ img[i+1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i+1][j]   ]++;
      GLCM[ img[i][j] ][ img[i+1][j+1] ]++;
   }
```

Figure 8. Pseudocode for GLCM

kernel highlights the overhead for indirect index calculation. The pseudocode is depicted in Figure 7.

6) Gray-Level Co-occurrence Matrices: Gray-Level Co-occurrence Matrices (GLCM) is a tabulation of how often different combinations of pixel brightness values (gray levels) occur in an image. Second order GLCM considers the relationship between groups of two (usually neighboring) pixels in the original image. It considers the relation between two pixels at a time, called the reference and the neighbor pixel. In this study, all 9 neighboring pixels are examined, as depicted in the pseudo-code in Figure 8. This kernel exposes the indirect index calculation overhead for matrix.

B. Large kernels

The large kernels are used to compare the performance of the SPE to the performance of the PPE. The chosen kernels are the Deblocking Filter of the H.264 video codec and the Viterbi decoder. These applications have a more regular behavior than the small kernels, mixing control and computation. It represents the impact of the SIMD overhead for scalar computation over an entire application.

1) Deblocking Filter: The H.264 Deblocking Filter (DF) improves the appearance of the decoded pictures by smoothing the block edges. The DF is highly adaptive and has different filter strengths depending on the block type. The DF kernel consists of almost 400 lines of C-code and 15 functions. First, it filters the left edges of the macroblock (MB) and then filters the vertical internal edges of its 16 4×4 blocks. This process is repeated for the horizontal edges.

The strength of the filter is determined dynamically and depends on the current quantizer, the coding of the neighboring blocks, and the gradient of the image samples across the boundary. There are five Boundary Strengths (BS) which the filter can apply, ranging from 0 (no filtering) to 4 (strongest one).

2) Viterbi Decoder: The Viterbi Algorithm [11] (VA) is an error-correction scheme for transmission through a noisy channel. It is used for decoding convolutional codes used in both CDMA and GSM digital cellular, dial-up modems, satellite, deep-space communications, and 802.11 wireless LANs. It is now also commonly used in speech recognition, keyword spotting, computational linguistics, and bioinformatics.

A formal description of the VA is that given a sequence Z of observations of a discrete-time finite-state Markov process in memoryless noise, the VA finds the state sequence X for which the posteriori probability $P(X/Z)$ is maximal, and thus it is optimal in that sense. Therefore, the VA is a solution to the problem of Maximum A Posteriori (MAP) estimation, which tracks the state of a stochastic process with a recursive method. The MAP sequence estimation problem is formally identical to the problem of finding the shortest route through a certain graph.

In this study the Zero-Tail (ZT) technique [12] is used. It consists of beginning the encoding with the contents of the shift register initialized to all zeros. The Zero Tail technique works as follow. For a positive integer L, we take as the code words in our block code all sequences of length $(L+m)n$ produced by inputting into the encoder a binary sequence of length L followed by m zeros. The resultant code is an $((L+m)n, L)$ block code of rate $(1/n)(L/(L+m)) = (1/n)(1-(m)/(L+m))$. The term $m/(L+m)$ is called the rate loss and is due to the zero tail.

V. Experimental Results

In this section, the experimental results are presented and analyzed.

A. Large-Data-Type

The LDT methodology highlights specific SPE characteristics that are known to introduce overhead. These kernels spend a significant part of their execution time on operations that require compiler-generated shuffles. Because of this, the effects of the overhead required to process scalar data can be measured. For analyzing the results the IBM cycle-accurate simulator was used as it provides kernels execution details. The simulator reports statistics such as the number of cycles spent by a specific area of the code, the number of single and double issued instruction cycles, the number of stalls, etc. This detailed information is necessary to evaluate all effects of the proposed methodology.

Figure 9 depicts the execution times of the LDT-emulated kernels normalized to the execution times of the original

kernels. It also breaks down the execution time into cycles spent on actual computation, NOPs, and stalls. The stalls are further divided into load/store (L/S) stalls, stalls waiting for the shuffle unit to become available, and stalls waiting for other functional units.

Overall, the LDT-emulated versions are 19% (merge sort) to 57% (FIR) more efficient than the original versions. This performance increase comes from several sources. Without the shuffle instructions, the kernels have less operations to perform and less stalls due structural hazards on the shuffle unit. This is represented by the decrease in computation cycles and by the elimination of shuffle stalls. The LDT methodology sometimes also affects the number of branch miss stalls and other stalls.

Analyzing only the variation in computation cycles, all emulated kernels reduce the number of cycles. For the sorting kernels, QuickSort and MergeSort, the number of computation cycles are reduced by 21% and 27%, respectively. SAXPY present a reduction of about 40% and the remaining kernels approximately 50%. The stall breakdown shows that the stalls on the shuffle pipeline are transferred to the Load/Store pipeline when going from the original to the emulated version of the kernels. On average, for the original versions, the stalls due to shuffle account for 19% of the total number of stalls, while the Load/Store stalls account for 12%. For the emulated versions, the Load/Store stalls account for 30% of the total number of stalls. The relative increase in the number of Load/Store stalls are because of two factors. First is the increase in performance of the kernel. As the kernel spends less time computing, the proportion of loads and stores increases. The second effect is the increased number of loads and stores. As data are not compacted in quardwords, there are more loads and stores.

While for the most kernels the percentage of branch misses does not change significantly, for the sorting kernels there is are a considerable variation. For QuickSort there is a decrease of 49% in the number of branch miss stalls while there is an increase of 37% for MergeSort. The compiler could increase the branch hints for the LDT version of QuickSort in comparison with the regular version. This explains the decrease in branch miss for that kernel. For MergeSort the opposite happened. The number of branch hint instructions decreased. In [13] a SIMDimized sorting algorithm for the Cell SPE is presented. Details of its performance are not presented, however.

Because of the small number of computations performed in each kernel, there are less than 36 live registers at any moment. For all kernels, except MergeSort, the number of used registers decreases in the emulated version. This decrease is the result of reducing the number of intermediate steps necessary to calculate the result. MergeSort has the same number of registers in both versions, FIR decreases the number of live registers by 14%, and the other kernels by 33%. These results show that the LDT methodology does not increase the pressure on the register file.

The results show the importance of scalar support for SIMD-only processors. The performance of every kernel increases when applying the Large-Data-Type methodology, which reveals the overhead of scalar processing on SIMD-only cores.

B. SPE-vs-PPE

In this section, the results for the SPE-vs-PPE methodology are presented and analyzed. Each kernel is executed 10 times and the first run is discarded as it is used to warm-up the cache. 10 runs are performed to minimize external influences on the execution times, such as operating system preemption. The average execution time of the late 9 runs is reported.

To evaluate the performance of the kernels, the real Cell processor has been used, because the IBM Simulator does not accurately model the PPE. Performance results are acquired using hardware counters. Hardware counters have a precision of approximately one microsecond. The execution times of the kernels are considerably larger than this precision.

1) Deblocking Filter: The input used for testing the Deblocking Filter consists of the first eight frames of the Lake Wave video sequence in the QVGA (320×240 pixels) resolution. Only the 8×8 upper left MBs are filtered as a larger area would not fit in the L1 data cache. The presented results are the average of the runs with eight frames each. As previously mentioned the PPE version only takes into consideration the measurements with the input data already present in L1 data cache. For the SPE version, it does not take into account the DMA transfer time.

The scalar version running on the SPE takes 300 μs to filter the frame area, on average. The scalar version running on the PPE is 12% faster, the required time to filter a frame area is 265 μs, on average.

The Deblocking Filter is an example of a well behaving multimedia kernel. The kernel operates on a predictable data set, which simplifies the data transfer to Local Store. The data processing loops have static boundaries, reducing loop overhead and making the branch hints more effective. The control loops and function calls do not incur additional overhead on the SPE. The compiler-managed overhead is dispersed by these operations, and therefore the overhead obtained with the SPE-vs-PPE methodology is smaller than that found with the LDT methodology.

In [14] a SIMDimized DF is presented with speedups of 2.6 over the scalar implementation. However, these results are obtained including the latency of memory accesses.

2) Viterbi Decoder: The execution time of the used Viterbi decoder is not dependent on the input data. Because of this characteristic and to avoid data transfer impact on performance, a constant input is used. The input is 100 blocks of 160 symbols long each.

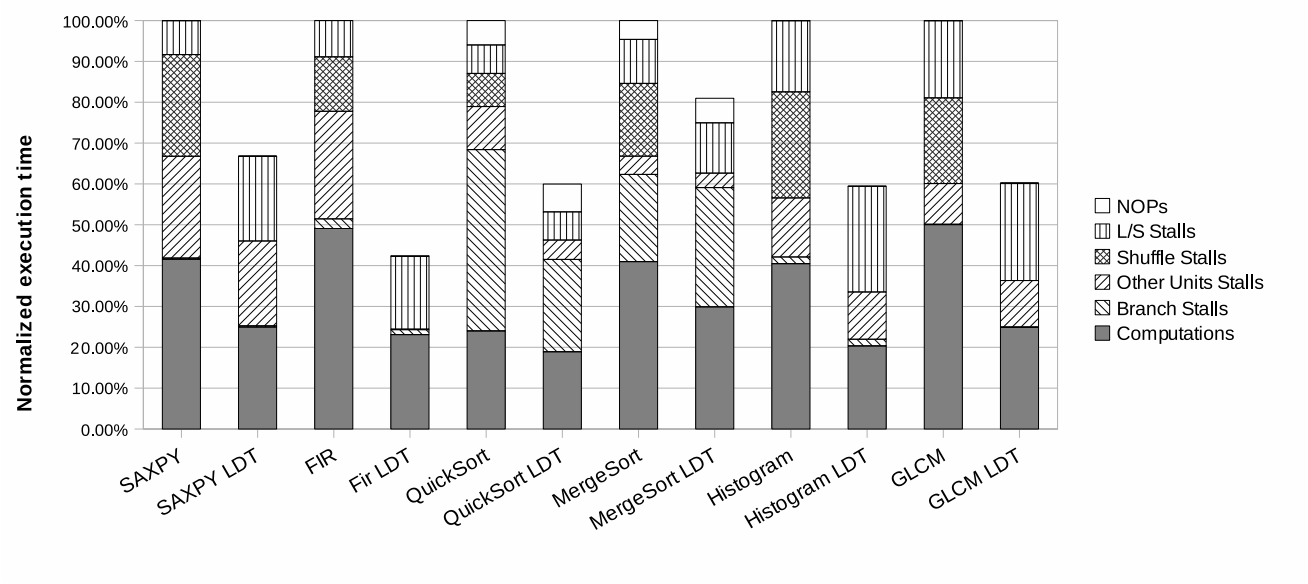

Figure 9. Execution times of the LDT-emulated kernels normalized to the execution times of the original kernels

The required computation time for the SPE was 10.6 ms while the PPE spent 6.5 ms for the same calculation. This result show that the PPU is 39% faster than the SPE for this application. This result is similar with the results found with the LDT methodology, because the Viterbi decoder is very computational intensive. In contrast to the DF, it spends most of its execution time in operations affected by compiler-managed scalar support overhead. The supported bandwidth for the PPU version is 2.45 Mb/s.

VI. Conclusions

In this paper the need for scalar processing support on SIMD-only architectures such as the Cell SPE is analyzed. Two methodologies are presented for this analysis, the Large-Data-Type and SPU-vs-PPE methodologies. The first simulates the behavior of a scalar unit by replacing all data types by the quadword data type, thereby eliminating the overhead required for processing scalars. This methodology can be used to evaluate small kernels. The second methodology compares the performance of the SPE to the performance of the PPE for large kernels.

The experimental results show that scalar support would provide considerable performance improvement for scalar data kernels. Results of the LDT methodology show that the scalar processing overhead on SIMD-only architectures ranges from 19% to 57%. For the considered large kernels, the Deblocking Filter and the Viterbi decoder, the overhead is 12% and 39The overhead is mainly caused by the additional shuffle instructions that need to be executed to move the scalar to the preferred slot and stalls waiting for the shuffle unit to become available.

To reduce the scalar processing overhead, a scalar unit could be added to the SPE. This, however, would require a complete re-engineering of the core including a separate register file and control logic to support another instruction set. The resulting area and power overhead goes against the reasons given in [15] for power efficient processors. Furthermore, adding a scalar unit would be beneficial only if the area increase would be less than the performance improvement. For example, if adding a scalar unit would double the area requirements, then it would be profitable only if single-core performance more than doubles the performance, since half as many cores can be placed on a single chip. Adding scalar instructions could also reduce power. Since the SIMD units are used to execute them, the unused parts can be turned off to reduce switching activity. It is reported in [9], however, that it was attempted during the development phase of the processor without consistent results.

To add support for scalar operations without sacrificing area, we propose special Load and Store instructions. These new instructions would load/store data directly into/from the preferred slot. The Cell SPE can load/store only quadwords. For the Load an extra layer of 4×1 multiplexers would be necessary to select the right word. The same applies to the Store instruction, but it would also require a masked write to the Local Store. With this masked write, only part of the quadword would be written to its final slot. With these additional instructions the performance of the Cell SPU for scalar and hard-to-SIMDimize applications would increase. We intend to evaluate these new instructions in future work.

ACKNOWLEDGMENT

This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6) and the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

REFERENCES

[1] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the Petaflop Era: the Architecture and Performance of Roadrunner," in *SC '08: Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[2] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 124–135, 1999.

[3] G. Ren, P. Wu, and D. Padua, "Optimizing Data Permutations for SIMD Devices," in *PLDI '06: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2006, pp. 118–131.

[4] D. Nuzman, I. Rosen, and A. Zaks, "Auto-Vectorization of Interleaved Data for SIMD," in *PLDI '06: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2006, pp. 132–143.

[5] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," *SIGPLAN Not.*, vol. 39, no. 6, pp. 82–93, 2004.

[6] A. Shahbahrami, B. Juurlink, D. Borodin, and S. Vassiliadis, "Avoiding Conversion and Rearrangement Overhead in SIMD Architectures," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 237–260, 2006.

[7] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE Inter. Symp. on*, 2007, pp. 62–71.

[8] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4, pp. 589–604, 2005.

[9] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[10] "Power Architecture Version 2.02." [Online]. Available: http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html

[11] A. Viterbi, "A Personal History of the Viterbi Algorithm," *Signal Processing Magazine, IEEE*, vol. 23, no. 4, pp. 120–142, 2006.

[12] H. Ma and J. Wolf, "On Tail Biting Convolutional Codes," *Communications, IEEE Transactions on*, vol. 34, no. 2, pp. 104–111, 1986.

[13] B. Gedik, R. R. Bordawekar, and P. S. Yu, "CellSort: High Performance Sorting on the Cell Processor," in *VLDB '07: Proc. of the 33rd Inter. Conf. on Very Large Data Bases*. VLDB Endowment, 2007, pp. 1286–1297.

[14] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of Video Filtering on the Cell Processor," in *Proc. of Inter. Symp. on Circuits and Systems (ISCAS)*, May 2008, pp. 488–491.

[15] H. Hofstee, "Power Efficient Processor Architecture and the Cell Processor," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th Inter. Symp. on*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–262.