Model-based Software Design Tools for the Cell Processor

Nicholas Lowell
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37203
lowellns@isis.vanderbilt.edu

ABSTRACT

This paper introduces the larger features of the Cell Processor that allow this specialized hardware architecture to provide a significant amount of increased performance. Specialized configurations call for specialized programming in order to harness the available performance increase. Such high computation configurations are prime targets for signal processing applications. There exists a tool set for modeling the dataflow of a signal processing application. A major goal exists to allow for generation of code to be used on the Cell. The first step involves learning the required techniques for programming by way of porting an example application to the Cell. This paper shows the first steps of utilizing the multi-core architecture which yields a significant increase in performance with room for further improvement in the future.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems, Signal processing systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User Interfaces*

General Terms

Design, Performance

Keywords

Cell processor, System dataflow modeling, Run-time kernel, Code generation, Automatic target recognition

1. INTRODUCTION

The slowing pace of decreasing transistor size and increasing clock speeds has diverted hardware developers to search for alternate configurations in which to increase computational power and speed. One such implementation is the production of multi-core architectures. The increase in number of processor units within a system gives way to physical parallel processing, yielding higher throughput than single-core architectures with both operating at similar speeds. A major difficulty in using such specialized hardware is the call for specialized programming, in both technique and the adoption of a hardware-specific API. Thus, adapting existing applications to these architectures either neglect to seize the computational advantage or require a significant amount of rewrite in order to realize the performance increase. The collaborative efforts from IBM, Sony, and Toshiba resulted in the fabrication of the Cell Broadband Engine Architecture (or simply the Cell). I have adopted this architecture and have ported an Automatic Target Recognition (ATR) application as the first steps in porting an existing signal processing tool set to take advantage of the specialized configuration and potential increased performance.

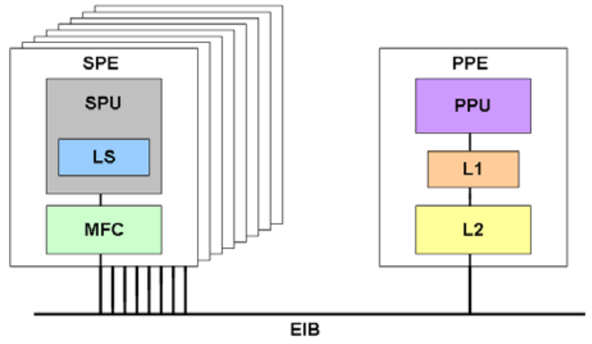

Figure 1: The Cell Broadband Engine Architecture

2. THE CELL PROCESSOR

The Cell Broadband Engine Architecture is mainly composed of a PowerPC Processing Element (PPE), eight Synergistic Processing Elements (SPEs), and an Element Interconnection Bus (EIB). A top level diagram of the architecture is in Figure 1.

2.1 The PPE

The PPE is a dual-threaded, 64-bit, big-endian, RISC processor that complies with the PowerPC architecture with Single-Instruction-Multiple-Data (SIMD)/vector extensions. It is composed mainly of a PowerPC processor unit (PPU) and a PowerPC processor storage subsystem (PPSS).

The PPU consists of separate 32 KB L1 instruction and data caches and six execution units for instruction execution. The PPSS handles memory requests from the PPU and memory-coherence from the EIB. Ports between the two components produce a capability for loading 32 bytes and storing 16 bytes independently per processor cycle. The PPSS has a unified 512 KB L2 instruction and data cache with a cache-line size of 128 bytes (same for the L1 caches). Notable registers for the PPE are 32 64-bit general purpose registers (GPRs), 32 64-bit floating-point registers, and 32 128-bit vector registers. Running at 3.2 GHz, the PPE is capable of its own intense processing, however, its main role is a system controller-running the operating system for the applications executing on the PPE and SPEs.[2].

2.2 The SPE

Each SPE is a 128-bit RISC processor with a new SIMD Synergistic Processing Unit Instruction Set specialized for data-rich and compute-intensive applications. Its composed of two main components-a synergistic processor unit (SPU) and a memory flow controller (MFC).

2.2.1 *The SPU*

The SPU has a 256 KB local store (LS) for both instruction and data, 128 128-bit GPRs, and no cache. It has four execution units, a direct memory access (DMA) interface, and a communication channel interface. Because a SPE does not have direct access to main memory, it must transfer any desirable data into its LS. It does this by sending DMA transfer requests to the MFC through the communication channel. The MFC then uses the DMA controller for the transfer.

2.2.2 *The MFC*

The MFC executes these DMA commands autonomously, allowing the SPE to continue execution during transfers. It can initiate up to 16 independent DMA transfers. The MFC serves as the SPU's sole interface to the external world: main-storage, system devices, and other processor elements. It handles communication (mailboxes and signal-notification messages) between the SPE and the PPE or other SPEs. Having two (odd and even) execution pipelines, the SPU is capable of completing two instructions per cycle, if the instruction types allow for it. It supports single-precision floating-point operations with a fully pipelined 4-way SIMD execution while double-precision operations are half pipelined. This includes combinational multiply-add operations for single precision, which means that two single-precision floating-point operations can be performed on four values per cycle, allowing for a *theoretical* performance at 3.2 GHz of 2 x 4 x 3.2 = 25.6 GFLOPS for each SPE.[2]

2.3 The Playstation 3

An easy source for obtaining the Cell processor is Sony's Playstation 3. With the supported functionality to partition the internal hard drive and install a Linux Operating System (Fedora Core 6 in this case), the privilege to develop on the Cell is readily available. One downside with using the Playstation 3 is the limitation to only six of the eight SPEs due to the setup of the system.

3. THE SIGNAL PROCESSING PLATFORM

There has been recent work to develop a Signal Processing Platform (SPP)[6] which supports model-based designing of high-performance signal processing applications. It consists of a modeling environment, a design-space exploration tool, analysis tools, a synthesis tool, and a heterogeneous dataflow execution platform. The work that is being presented here mainly dealt with the execution platform.

The synthesis tools map a signal processing model to this dataflow execution platform. It includes a light-weight real-time non-preemptive kernel and is associated with a build tool that can synthesize necessary glue and configuration artifacts such as communication maps, schedule tables, and interface specifications for the platform. If the target hardware calls for it, these generations (such as the kernel) come in the form of C source files that can be compiled with a set of Makefiles included with the execution platform. The first step for the current work was to modify the Makefiles and light-weight kernel so that it could compile and execute signal processing applications in a concurrent manner in a Linux environment, specifically: the Cell processor in the Playstation 3 running Fedora Core 6.

4. THE CELL-SUPPORTED SPP EXECUTION ENVIRONMENT

The Makefiles for compiling the various environment support packages now include rules for the Cell that link the SPP makefiles with those provided with IBM's Cell Software Development Kit (SDK)[4, 5] by way of some target addition, variable renaming, and special scripting to force the SDK makefiles to produce output in a file structure that matches the results if one was to compile with, say, Microsoft Visual Studio on an ix86 machine.

Previously, the run-time kernel scheduled and ran the signal processing blocks in a round-robin, sequential manner. Now, to support developing applications for the Cell, the kernel supports concurrency. It does this by creating a POSIX thread (pthread) upon the first execution of each process. These threads run the signal processing blocks, which now require a simple infinite loop that wraps the execution code. This proved to be a wise decision because when tasks are loaded and run on SPEs (these tasks are called *contexts*), this is done with a blocking function call. Therefore, this thread setup is the only way to run multiple contexts.

Lastly, in order for data to safely flow through the concurrent system, the communication streams between the threads are now synchronized and treated as critical resources. Mutexes nicely control the access to the interprocess communication streams and produce thread stalls upon denied requests, as opposed to wasteful polling loops. With the updated execution environment, I present a sample signal processing application developed for the Cell.

5. AUTOMATIC TARGET RECOGNITION EXAMPLE

The example system used for monitoring the adaption of the SPP kernel is an embedded real-time Automatic Target Recognition (ATR) system as taken from [1], whose dataflow model is in Figure 2. This image-processing system finds and classifies the objects in the input images that belong to a set of target classes. Main processing involves correlation filtering, where each image is correlated with template filter images associated with different target classes, producing a set of peak locations that pinpoint potential targets. These locations form the center of regions of interest (ROIs) that

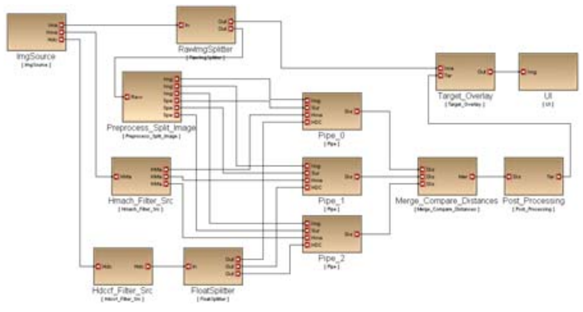

Figure 2: Dataflow Model of ATR

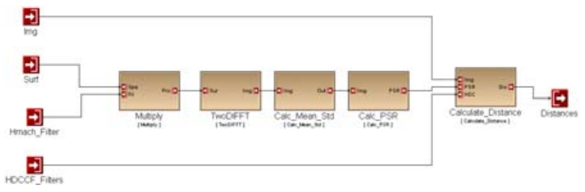

Figure 3: Internal Structure of Pipe in ATR

are processed further and identified using a Distance Classifier Correlation Filtering (DCCF) algorithm, which increase the confidence level of the identified peaks.

The example system uses images of 128x128 resolution and can extract up to eight ROIs (of size 32x32) while searching for three possible target classes. This computation-intensive ATR algorithm involves a pre-processing step involving a 2D FFT on the image in order to correlate the images in the frequency domain. Multiple copies of the current image pass through one computational pipe, as modeled in Figure 3, for each possible target class with which the image can be correlated. A Hmach filter provides these target classes. Before leaving the pipe, the distance to the other target classes are calculated according to the DCCF algorithm. Finally, the locations of the targets (if any) are located in the image after the outputs from the pipelines are merged and post-processed.

5.1 Adapting Algorithm Kernels for the Cell

With the new SPP kernel in place, producing threads for each block and using synchronized communication queues, an application that is functional on another supported platform, like an ix86 PC, requires minimal modification to simply achieve proper execution (I will worry about performance later). The main update is each task's source code is wrapped in an infinite loop, since now it will only be called one time as a thread and expected to run continuously and concurrently with the other tasks. It is also very important to remember that the PPE is a 64-bit, big-endian core. Thus, additional changes to the algorithm dealt with the storage model of the data. It is impossible to give universal specifics for adapting any application because these types of changes are application specific. In the case of the ATR, the byte order of the input files (the images, Hmach filter data, and DCCF filter data) have to be reversed for proper reads. Furthermore, the ATR performs queue size checks for zero instead of depending on a null return from dequeuing on an empty stream, which no longer occurs but now stalls. Compiling the updated kernel and ATR project, I was able to successfully run the application on the PPE of the Cell processor. Next, I present more adjustments to the ATR in order to move the computation-heavy tasks to the SPEs. In preparation for this step, deeper analysis of the system was necessary.

5.2 Memory Consumption

It was important to look at the amount of memory required by the system at the high computation moments (the pipelines of the system) because the necessary resources for the computational steps must fit onto an SPEs LS along with the corresponding code. Walking through the pipeline blocks from Figure 3, "Multiply" receives an image and the Hmach filter data. Each pixel of an image is stored as a complex (real and imaginary parts) value as a single-precision floating-point type (size of four bytes for the Cell). Therefore, with an image of size 128x128, it consumes 128*(128*2)*4 = 128 KB of memory-half an SPE's LS! What's worse is that the Hmach filter data is the same size. These two data structures alone would completely consume an SPE's 256 KB LS, leaving no room for code and other miniscule data.

Copies of the processed image continue through the pipeline to the 2D-IFFT, "Calc_Mean_Std", and "Calc_PSR", none of these tasks really needing much more memory for large data other than the image. The traveling image actually ends its journey in the "Calc_PSR" task which sends out peak-to-sideloab ratios for the detected peaks (up to eight peaks).

The final block in the pipe, "Calculate_Distance", is a rather hefty task. In order to extract and normalize the ROIs and calculate their distances (which involve multiple FFTs and IFFTs among other steps), the task calls for a copy of the original image, a place to store the ROI, another area for storing intermediate results, and the Hdccf filter data. Fortunately, this filter data is only 8 KB rather than the 128 KB size of the Hmach filter data. Unfortunately, the image, the ROI, and the buffer are 128 KB each. The amount of data sums to more than 384 KB-far exceeding that of an SPE's LS-and the larger-than-average code still must be included.

Being able to move these computation intensive tasks to the SPEs required an analysis and search for the truly essential resources as well as potential for segmented processing steps. Furthermore, with the initial design, with three pipes, each consisting of five blocks, there are 3*5 = 15 individual tasks and only six SPEs. The methods used for eliminating these dilemmas are discussed in the next section.

5.3 Algorithm Analysis

In order to fit the high-computation pipelines of the ATR application onto the six available SPEs, a deep analysis of the inner workings of the tasks and the detailed flow of the involved data was required. The two major obstacles to be handled were *task allocation*-how the several task blocks should be placed on the SPEs-and *resource management*-how the required data could fit onto the limited LSs along with code

5.3.1 *Task Allocation*

For task allocation, the apparent solution to fitting three pipes (of five tasks each) onto six SPEs was providing two SPEs per pipe and actually merging the five tasks in a pipe into two larger tasks. The desired result of these mergers would be two tasks of an equal computational load. Be-

Table 1: Run Time and FLOP Count of Pipe Tasks

Task	Run Time (μsec)	FLOP Count
Multiply	984	193548
TwoDIFFT	4490	1179648
Calc_Mean_Std	261	49159
Calc_PSR	1326	195440
Calculate_Distance	7889	1948448

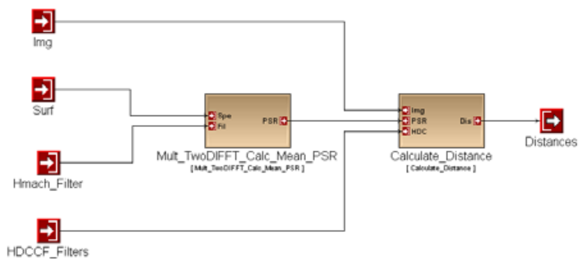

Figure 4: Structure of Merged Pipe in ATR

cause of the synchronized queues employed in the system, if one task executes longer than the other, the shorter task will end up stalling as it waits for data to be pulled from the interconnecting queue. Analysis of both the execution times (on the PPE) and number of floating-point operations (FLOPs or FLOP count) for each block provided an idea for the best division of labor. The values are listed in Table 1.

Based on these values, if the first four tasks were merged into a single task, it would theoretically execute 193548 + 1179648 + 49159 + 195440 = 1617795 floating-point operations in 984 + 4490 + 261 + 1326 = 7061 microseconds. The first four tasks combined are still not as hefty as the "Calculate_Distance" task. However, it is the best attempt to balance the load. Thus, the pipes were redesigned to contain only two task blocks, as seen in Figure 4, and the corresponding source files for the (originally) first four tasks were glued together into one task, called "Mult_TwoDIFFT_Calc_Mean_PSR". This satisfied the task allocation requirements for porting to the SPEs.

5.3.2 Resource Management

With sufficient task allocation, the more difficult job of resource management required a deep look into what data was required and where in the two tasks. Because of the modularity of the system, the two task blocks can be viewed separately, starting with "Mult_TwoDIFFT_Calc_ Mean_PSR." Note that though this is one source and one task, the four components it is composed of remain separated within the code. Therefore, the single task or the four separate tasks of which it is composed may be referred to interchangeably.

Recall that the large data structures existent in the first four tasks are the image and the Hmach filter data, both 128 KB each. Furthermore, originally a copy of the progressively processed image was passed through each task because they were threads operating on an image concurrently with the other tasks, actually consuming 128 * 4 = 512 KB. Obviously, this is impossible for the SPE, and fortunately, this is no longer required because these four tasks now operate sequentially. Thus, a single image can be passed through the algorithm and data processing is in place. However, there is still 256 KB of data. Because the image travels through a majority of the task, it is preferable to keep this structure in the LS. Looking at the Hmach filter, it is realized this data structure is only necessary during the "Multiply" processing, the first part of the task. Furthermore, there is a one-to-one relationship between the filter values and the image values. This situation allowed me to use part of the Hmach filter data on part of the image and then write over with another part of the filter data for processing the next corresponding part of the image. It is sufficient to store half the Hmach filter (64 KB) at a time, resulting in data storage size of about 192 KB, leaving 64 KB for code (which was also sufficient).

As mentioned earlier, the lengthy "Calculate_Distance" task originally required over 384 KB of data: the image, a ROI image structure, and an intermediate ROI buffer. This task's algorithm involves acquiring a ROI from a section of the image, and through several stages of computation (where the intermediate ROI buffer is required) the distances are calculated. This is repeated eight times, writing a different ROI from the image each time. Because of the repeated use of the image, it is desired to once again retain it in the LS. Furthermore, recall that a ROI has a resolution of only 32x32. The full image structure is used because it matches the necessary structure for the ROI, however, the ROI uses 1/16th of the structure. The necessary size for an ROI structure is only 32*(32*2)*4 = 8 KB, significantly smaller. By implementing this new data structure for the ROI and the intermediate ROI buffer, the amount of data required for the "Calculate_Distance" was reduced to about 144 KB, leaving plenty of room on the LS for the other necessary components (i.e. code). With the methods for task allocation and resource management fully defined, I present the performance achieved from running the ATR on the Cell, looking at before and after SPE support.

5.4 Preliminary Performance Analysis

I first provide a performance point of reference of the system when running solely on the PPE of the Cell. Then the steps taken to implement the pipe tasks to the SPEs will be detailed, followed by the complete implementation's performance results.

In order to have a feel for the improvement achieved when the ATR system makes use of the SPEs, performance of the "before" setup needed to be acquired. Data in terms of millions of floating-point operations per second (MFLOPS), throughput of data, and even frame rate (due to the image-processing nature of this example) were calculated by measuring the run times for processing 1000 images and taking the average. Knowing the number of floating-point operations (FLOPs) per image and the size of an image allowed me to compute the various performance parameters. The breakdown of FLOPs for each block in Figure 2 (with the pipes expanded to show the two internal blocks) is shown in Table 2.

The ATR with the now two-task pipes were run on the PPE of the Cell in the traditional serial manner with the old kernel (and some slight hacking) and also with the tasks looping within pthreads, labeled as "Serial" and "Pthreads", respectively. The resulting average run-times for processing a single image on the PPE and the calculated performance parameters are found in Table 3.

Though it may be considered minor, the mere implementation of concurrency provided some improvement over the original behavior of the kernel. Taking advantage of the

Table 2: FLOP Count for ATR Tasks

Task	**FLOP Count**
ImgSource	0
Hmach_Filter_Src	0
Hdccf_Filter_Src	0
RawImgSplitter	0
PreProcess_Split_Image	1146880
FloatSplitter	0
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Calculate_Distance	1948448
Calculate_Distance	1948448
Calculate_Distance	1948448
Merge_Compare_Distances	0
Post_Processing	47
Target_Overlay	0
UI	0
TOTAL FLOP COUNT	**11845656**

Table 3: Reference Pre-SPE Performance

FLOP Count	**11845656**	
Image Size (bits)	**1048576**	
Task	**Serial**	**Pthreads**
Avg Run Time (msec)	46.40	40.01
MFLOPs	255.29	296.05
Throughput (Mbits/s)	22.60	26.21
Framerate (frames/s)	21.55	25.00

powerful SPEs greatly increases this performance. With these foundational performance bases in place, I now cover the process of moving the computational pipes to the SPEs and then present the performance results.

5.5 Using the SPEs

Originally, each task's computational core was surrounded with the necessary data acquisitions and preparations, mainly consisting of the retrieval and placement of the data from and onto the communication queues, respectively. The PPE tasks continue to perform these jobs. However, now the computational cores of the "Mult_TwoDIFFT_Calc_Mean_PSR" and "Calculate_Distance" tasks reside on the SPEs. The corresponding PPE tasks have the added responsibility of setting up and running the SPE contexts–passing in the effective addresses of the data required by the SPEs. Upon running the SPE context, the PPE simply blocks until the SPE has completed processing. This is the chosen method for letting the PPE-side know when the data has been processed and can be added to the output queue.

Along with the computational cores now SPE code, all pertinent data must be moved from main memory to the SPE's LS during execution. Analogous to the PPE data preparation steps, the SPEs perform DMA operations, with a 16-byte alignment (optimally 128-byte aligned), before processing to acquire the necessary data and afterwards to place the newly processed data back into main memory for the PPE to pass through the system. With these wrapping instructions in place, the only remaining step to building our SPE-supported ATR application was implementing the

Table 4: Performance Comparisons

FLOP Count	**11845656**		
Image Size (bits)	**1048576**		
Task	**Serial**	**Pthreads**	**SPEs**
Avg Run Time (msec)	46.40	40.01	10.09
MFLOPs	255.29	296.05	1174.31
Throughput (Mbits/s)	22.60	26.21	103.95
Framerate (frames/s)	21.55	25.00	99.13

changes to the computational cores discussed earlier.

Now, the algorithm for the "Mult_TwoDIFFT_Calc_Mean_PSR" diverges when it comes to using the Hmach filter data. The filter is used for complex conjugate multiplication with the image data, in a one-to-one data entry correspondence. The necessary resource management of using half the Hmach filter at a time is implemented through a DMA of the first half of the data and using it in conjunction with the first half of the image. After returning, a second DMA places the second half of the filter data in the same LS location as the first half. Then the multiplying core executes again, with an added parameter indicating the second call to this function, telling it to use the second half of the image this time. In summary, originally a single call to this multiply core would carry out the processing for all 128 rows of the image, but this segmentation now has it performing on only 64 rows at time (either the first half or second), requiring a second call. Following this step, the remaining computational cores are called as usual, the only difference being the elimination of any copying of the image but instead using the same location throughout the entire task.

After adding the reduced-size ROI data structure to the system, the only necessary changes to the "Calculate_Distance" computational core were replacing any instances of expecting an Image structure with the ROI structure. The task performs successfully with this setup. With the computational cores of the pipes of the ATR system fully implemented on the six SPEs, the performance of the system are presented next.

5.6 Final Performance Analysis

Table 4 contains the performance references presented earlier along with the measured performance of the ATR system that takes advantage of the SPEs, named "SPEs." As is shown, using the SPEs of the Cell increases the performance of the ATR system by 300% over simply using pthreads and more than 350% over the original sequential execution. A breakdown of the run times for the SPEs tasks revealed "Mult_TwoDIFFT_Calc_Mean_PSR" taking about 7.5 milliseconds while "Calculate_Distance" completed after about 10 milliseconds. When compared to the complete run times of the system (about 10 milliseconds), notice that "Calculate_Distance", the most computationally intensive task of the system, is the delaying factor for the system. Nevertheless, this successful model development and implementation of the ATR system onto the Cell processor within the Playstation 3, making use of the six SPEs, has proven to be a significant achievement in improved performance and, in general, in the work involving the SPP. Future steps in this work are discussed next.

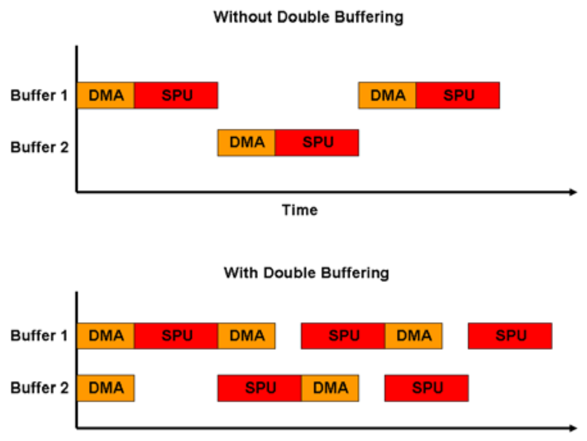

Figure 5: Benefits of Double Buffering

6. FUTURE OPTIMIZATIONS

Even while achieving more than one billion FLOPS (GFLOPS) with the initial porting of the ATR system to the Cell, there are a number of optimization techniques that remain unimplemented. Because the speed of the SPE tasks (specifically, “Calculate_Distance”) are the major factors in the completion time, the focus should be spent on optimizing those tasks.

Such possible optimizations, as mentioned in [3], are methods such as double buffering the DMA operations. Instances where large amounts of data are brought into the LS through DMA operations and then used for processing could be broken up into smaller transfers with segmented processing intertwined. Because the DMA operations are handled autonomously by the MFC, it is wasteful to wait for these operations to complete before performing any computations. Therefore, a DMA operation should commence on future data before processing the current data begins. Figure 5 shows a comparison of general operations with and without double buffering.

The current algorithms for the SPEs task do not take advantage of the SIMD capabilities of the SPUs. The use of vectors would allow for operations on four single-precision floating-points at a time, as opposed to single values in the current model. This implementation would require significant modifications to the code of the two tasks, however, performance has the potential to increase four-fold.

Furthermore, loop unrolling has been suggested as a method for squeezing extra performance out of the SPEs. This method provides more instructions for the SPU, allowing for more instruction interleaving and helping to prevent any stalls that may occur.

Some more general methods for performance improvement are things such as looking for redundancy, such as if DMA operations do not need to run every loop. Perhaps there are instances where the effects of regenerating some data versus copying (or moving with DMA) need to be analyzed. It is important to find generalized methods for optimizations as development moves away from the specific ATR example and into the general modeling and generation from the SPP.

7. FUTURE WORK AND CONCLUSION

The experience of adapting the ATR example to the Cell processor has guided us through modifying the SPP kernel that is associated with its generation tools. Furthermore, it has provided a look into the techniques needed to run any type of signal processing system that can be modeled with the SPP tools. One in particular is the merging of blocks for proper SPE scheduling. Making this an automated feature in the modeling environment is a strongly desired featured. Therefore, a major future task involves incorporating this and other techniques directly into the tool suite, allowing for the generation of these techniques based on the modeled system and eliminating the need for manual adaption.

8. REFERENCES

[1] S. Asaad, T. Bapty, and S. Neema. Performance modeling for adaptive parallel embedded systems. In *IEEE International Performance, Computing, and Communications Conference Proceedings*, pages 57–64, April 2002.

[2] IBM. *Cell Broadband Engine Programming Handbook*, April 2007.

[3] IBM. *Cell Broadband Engine Programming Tutorial*, March 2007.

[4] IBM. *Cell Broadband Engine SDK Libraries Overview and Users Guide*, March 2007.

[5] IBM. *Software Development Kit 2.1 Installation Guide*, March 2007.

[6] S. Neema, T. Bapty, J. Scott, and B. Eames. Signal processing platform: a tool chain for designing high performance signal processing applications. In *IEEE SoutheastCon Conference Proceedings*, pages 302–307, April 2005.