MODEL-BASED SOFTWARE DESIGN TOOLS FOR THE CELL PROCESSOR

By

Nicholas Stephen Lowell

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2009

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sandeep Neema

To my

ever caring, ever supportive,

ever encouraging,

ever humbling, ever faithful,

and ever loving

parents.

ACKNOWLEDGEMENTS

This research was performed under the support of a Raytheon grant and Raytheon contact, Andrew M. Vandivort.

I want to thank Dr. Sandeep Neema, Dr. Gabor Karsai, and the Institute for Software Integrated Systems (ISIS) for the wonderful support, guidance, and opportunity to further my education and experience at Vanderbilt University and as a part of ISIS.

I also want to thank those professors at Lipscomb University who willingly went beyond the call of duty to teach me and prepare me for all of life's opportunities and challenges.

Finally, I thank my friends and my family. They bring me joy and laughter, they exemplify faithfulness and love, and they believe in me even when I do not.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API — Application Programming Interface

ATR — Automatic Target Recognition

CPU — Central Processing Unit

DCCF — Distance Classifier Correlation Filter

DMA — Direct Memory Access

DSML — Domain-Specific Modeling Language

DSP — Digital Signal Processor

EA — Effective Address

EIB — Element Interconnect Bus

FFT — Fast Fourier Transform

FLOP — Floating-Point Operation

FLOPS — Floating-Point Operations Per Second

FPGA — Field Programmable Gate Array

GFLOPS — GigaFLOPS

GME — Generic Modeling Environment

GPR — General Purpose Register

GUI — Graphical User Interface

IFFT — Inverse Fast Fourier Transform

IO — Input/Ouput

KB — Kilobyte

LS — Local Store

MACH — Maximum Average Correlation Height

MFC — Memory Flow Controller

MFLOPS — MegaFLOPS

OS — Operating System

PC — Personal Computer

PPE — PowerPC Processing Element

PPSS — PowerPC Processor Storage Subsystem

PPU — PowerPC Processing Unit

PSR — Peak-to-Sideloab Ratio

RISC — Reduced Instruction Set Computing

ROI — Region of Interest

SDK — Software Development Kit

SIMD — Single Instruction-Multiple Data

SPE — Synergistic Processing Element

SPML — Signal Processing Modeling Language

SPP — Signal Processing Platform

SPU — Synergistic Processing Unit

UI — User Interface

CHAPTER I

INTRODUCTION

As improvements slow in transistor size and clock speed, hardware developers 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. 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 that supports the non-traditional instructions. Thus, adapting existing applications to these architectures either neglects to seize the computational advantage or requires a significant amount of code rewrite in order to realize the performance increase.

As the complexity of embedded systems grows, the traditional method of manual code production from start to finish becomes too long and costly for practicality. The use of domain-specific modeling languages (DSMLs) [12] allows designers to describe a system using semantics appropriate for that system. Accompanying the modeling language are automation tools for transforming or generating the modeled system into a physical product (i.e. source code) in a fraction of the time it would take a user to develop the same system writing only code from start to finish. The Generic Modeling Environment (GME) [15] is a

toolkit for defining such DSMLs and assists in building the tools (i.e. model transformers, code generators) one desires to accompany the language.

I have adopted the Cell Broadband Engine Architecture (or, Cell)—the multi-core architecture resulting from the collaborative efforts of IBM, Sony, and Toshiba—and have adapted a DSML tool suite, the Signal Processing Platform (SPP) [17], to take advantage of the multi-core structure and improve signal processing performance while requiring minimal extra work from the user by handling most of the specialized programming.

The second chapter provides some general background information about multi-core architectures and some related work. The third chapter presents preliminary information surrounding the Cell architecture. Chapter four introduces the SPP tool chain—the modeling language and the associated tools—and highlights the updated features that support the Cell and take advantage of its multiple cores. Following, the fifth chapter provides an example Automatic Target Recognition (ATR) application showing how to port from a typical PC CPU to the Cell using the updated tools. Chapter six contains the results of the example, comparing performances between a typical PC chip and the Cell. Finally, chapter seven previews the future progressions of this project while chapter eight presents conclusions drawn from this work.

CHAPTER II

BACKGROUND

A multi-core processor is an arrangement of multiple independent processing cores on a single chip, typically in a manner in which some resources are shared among the cores. The architecture may require the cores to share as much as cache, memory, and busses, or the cores may have a subset of individual components that eventually diverge to a shared source. Figure 1 shows different possible core architecture designs.

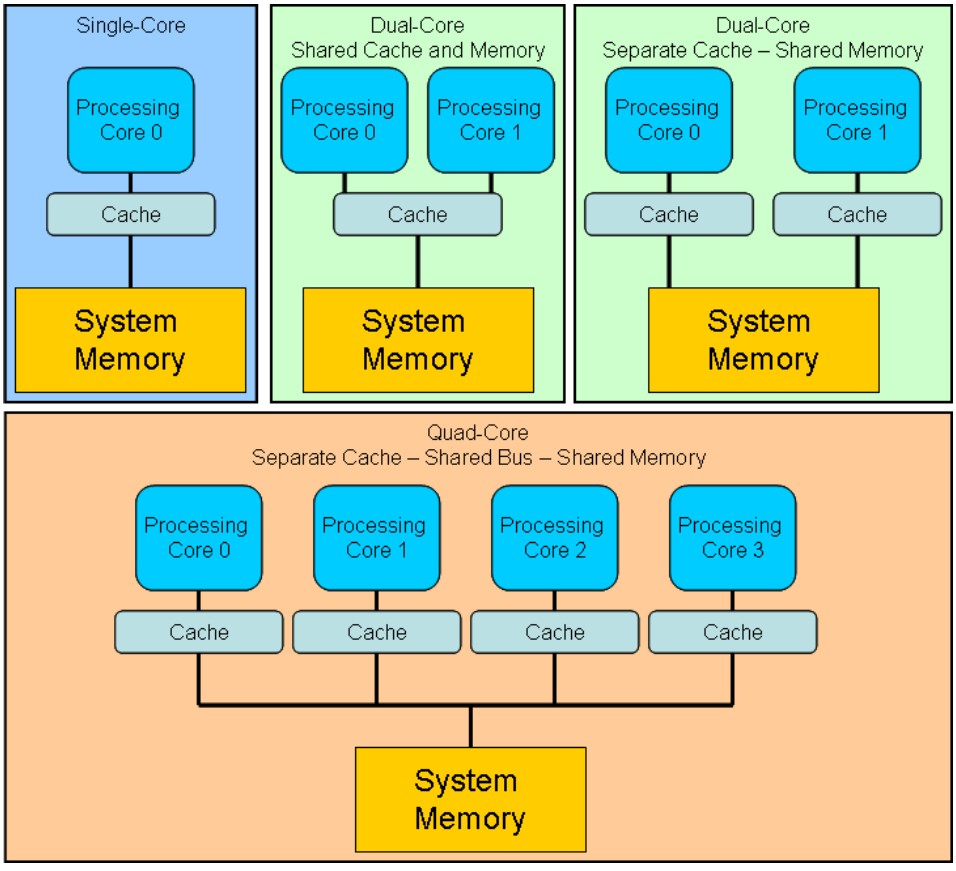


Figure 1. Different Processor Architectures

Regardless, there exist physically independent processing units that can be used to share the workload of a pressing application, allowing for significant improvements in various aspects of the system, such as execution performance and physical demands. However, the newness and complexity of the arrangement call for new implementations of applications and embedded systems—thinking about division of labor and proper balance. A slightly different issue sprung from the multi-core emergence involves moving preexisting single-core applications to the architecture and maintaining the temporal and reliability constraints often present in embedded systems. Work exists in both areas as developers strive to adapt to the changing hardware environments and reap the benefits.

There are both physical and execution reasons for using multi-core architectures. Early on, as chip developers increased their transistor density for faster processor clocks, the combination of higher voltage requirements and increased current leakage increased the amount of power usage. This also increased density began to produce denser and faster heat production. As clock speeds increased, builders were unable to keep up with dissipating the heat produced by the high-speed single core. The solution for increased performance without burning chips involved going around the heat issue by adding more cores to a single chip—cores that run at slower clock speeds—thus allowing for efficient heat dissipation and, additionally, reducing the power requirements of the chip [20]. The available advantage, then, of multi-core processors that a user can utilize is physical parallelization. Developing applications to use all present

independent CPUs simultaneously is a sure-fire way to reach the system's goal faster.

As it has already been alluded to, the key to developing for multi-core architectures involves paralleled systems. The most prominent method involves identifying the parallelism and implementing multiple threads to execute these tasks simultaneously [20]. However, as [20] and [4] point out, several issues must be addressed to ensure proper concurrent execution. These are issues such as preventing opportunities for race conditions and deadlocks and handling communication between the threads. [4] also mentions how the performance increase will be reduced due to necessary overhead that handles work distribution, and it warns of a thread threshold in some systems where the overhead consumes more time than the computational cores of the threads, and the performance depletes. This alludes to a difficulty in developing scalable code—code that will not decrease in performance if the number of cores increases. One problem with developed single-core applications is that they often lack this feature, making it difficult to move to a multi-core architecture. Porting such an application requires extensive analysis from the user or an almost complete redevelopment.

It is perhaps more difficult to take preexisting single-core specific applications and port them to a multi-core layout while maintaining the integrity of the system. [19] proposes a method for porting a large embedded system to a multi-core processor that involves a two step process of componentization and then partitioning of the tasks of the system. Components are logically grouped

tasks from a design point of view and independent of the architecture. Partitions, whose groupings may intersect with components, are arranged based on their ability to run concurrently and on which core; thus, they are dependent on the architecture. The work to be presented develops slightly different ideas of a component and the partitioning is related to that grouping, as opposed to the orthogonality in [19]. Whether porting existing systems or developing new applications, [22] and [16] talk about this difficulty and the necessary analysis of shared resources and task allocation. It is important to identify the bottlenecks in the program and avoid them if possible, and to test multiple methods that are algorithm-friendly. In talking about the impact of network sharing in multi-core architectures, [16] reveals how different work allocation methods improve different types of systems and can hinder a system if the wrong one is chosen, producing unnecessary dependencies and latency.

When it comes to performance analysis, [13] looks at key points for benchmarking multi-core architectures in order to find the sources of latency in order to focus performance tuning in those locations. The two main sources of inefficiencies were memory and cache. As will be mentioned, some of the cores on the Cell processor actually lack cache—eliminating that source of latency and improving benchmark simplicity. [4] takes time to address the difficulty to debug parallel programs, especially using sequential debuggers.

Some specific areas in which applications are being optimized for multi-core architectures to achieve increased performance are [22] which talks about network security applications and [2] which focuses on optimizing an FFT for a

multi-core architecture. It highlights the importance of domain specific knowledge and analyzing important issues of the specific application such as load balancing, work distribution, and data-reuse. The work to be presented parallels this but extends it to a more abstract perspective.

Something many of these works have in common is the assumption of identical cores. With this, some of them place a great deal of the task distribution on the shoulders of the Operating System. The work to be presented here greatly differs in this venue. As will be discussed, the Cell processor has two distinguishable core designs and the task allocation is entirely up to the developer and is more involved due to a limitation of available resources.

CHAPTER III

CELL PROCESSOR

The Cell Broadband Engine Architecture is mainly composed of a PowerPC Processing Element (PPE), eight Synergistic Processing Elements (SPEs), and an Element Interconnect Bus (EIB). A top-level diagram of the architecture is in Figure 2 with the die in Figure 3.

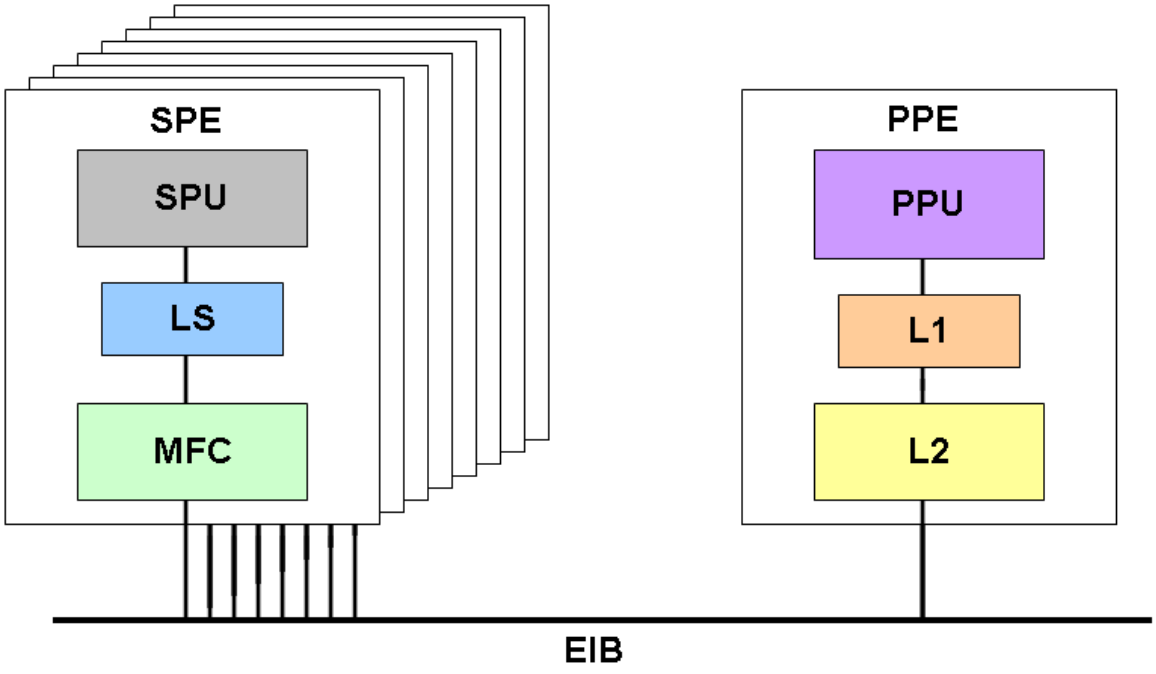


Figure 2. The Cell Broadband Engine Architecture

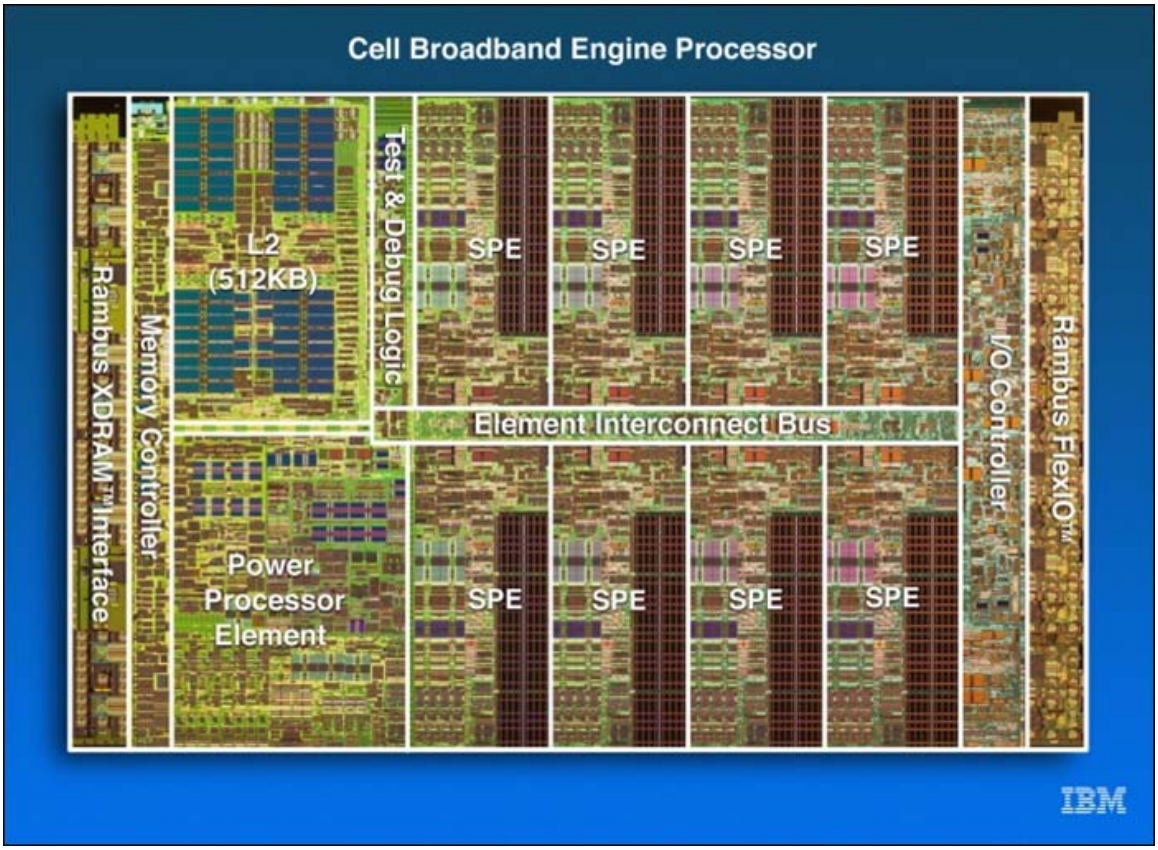


Figure 3. Die of Cell Processor[1] [10]

PowerPC Processing Element

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), L1 cache, and a PowerPC processor storage subsystem (PPSS).

The PPE contains 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

[1] Reprint Courtesy of International Business Machines Corporation, copyright 2009 © International Business Machines Corporation

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. [5]

Synergistic Processing Element

Each SPE is a 128-bit RISC processor with a new SIMD Synergistic Processing Unit Instruction Set specialized for data-rich and computationally heavy applications. It is composed of three main components—a synergistic processor unit (SPU), local memory store, and a memory flow controller (MFC).

The SPE 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. It is important to note that the effective address of the data to be moved must be minimally quad-word aligned (with the optimal being 128-byte aligned).

Furthermore, DMA transfers can be in sizes of 1, 2, 4, 8, or a multiple of 16 bytes with a limitation of 16 KB per transfer.

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 $2 \times 4 \times 3.2 = 25.6$ GFLOPS for each SPE. [5] It is important to highlight this as the theoretical maximum performance. Most pertinent applications involve instructions other than floating point multiply-adds and so practical performance is decreased.

Memory Flow Controller

The MFC executes DMA commands autonomously, allowing the SPU to continue execution during transfers. It can initiate up to 16 independent DMA transfers. The MFC serves as the SPE's sole interface to the external world: main-storage, system devices, and other processor elements. It also handles communication (mailboxes and signal-notification messages) between the SPE and the PPE or other SPEs.

Element Interconnect Bus

The EIB is the connection between all components of the Cell—the processors, main memory, and IO controllers. It has four unidirectional 16-byte data rings (two clockwise, two counterclockwise) that transfer 128 bytes at a time. Processors are capable of sending and receiving data simultaneously. Its internal maximum bandwidth is 96 bytes per cycle and it can support over 100 outstanding DMA requests between main memory and the SPEs. Saturating the high bandwidth bus would be a hard task for most practical applications.[5][6][14]

Sony 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 [21] with kernel-2.6.21 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. Sony disables one SPE for increased yields and the other SPE is used by a virtualization layer (called the hypervisor) that falls between the Linux kernel and the physical layer [14]. Cheap access to the only slightly limited Cell is acceptable enough for development purposes.

CHAPTER IV

SIGNAL PROCESSING PLATFORM

Past work has developed a Signal Processing Platform (SPP) [17] which supports model-based design 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 following sections briefly describe the features and focus on the added functionality that allows for supporting the Cell.

Signal Processing Modeling Language

SPML is a defined DSML, integrated into GME, with the functionality for modeling the dataflow of a signal processing system (or application). There are four main features of the modeling environment, of which Appendices A-D show their defined meta-models. The first is the ability to define data types relevant to the system, which are then used in collaboration with defined components. Components are individual blocks that are designated to perform one task of the system (i.e. filter, FFT, correlation, etc.). Later, each component comes to represent one process block during execution. Often included in a component are input and output ports for designating entry and exit points for the data and a core block for representing the computational task itself. The data types that are defined are used for associations with the ports of the component. The third

feature of SPML is a hardware definition platform. Basic architectures and hardware profiles can be created and indicate the types of available communication protocols, the device type (CPU, FPGA, memory device, DSP), and a few details that pertain to that specific device (such as the type of CPU). Finally, SPML provides the venue for arranging and connecting the defined components to build the desired system's dataflow and appoint a hardware platform to the components. [17]

To support the Cell processor in the SPP, it needed to be added to the available hardware types. Though it is a multi-core architecture, the decision was made to have it fall into the CPU category. Users would be able to define a CPU hardware type and then designate that CPU as a CELL. The tools associated with the SPP see this selection and know how to treat the architecture.

Execution Platform

The dataflow execution platform of the SPP includes a lightweight real-time non-preemptive kernel. Modeled in Figure 4, the kernel handles the management and movement of the data throughout the system from process block to process block. The process blocks interact with the kernel using a simple API for actions such as acquiring and returning memory buffers, bringing in the next set of data (called *dequeuing*), or pushing out processed data (called *enqueuing*). The benefit of the kernel handling the dataflow is that the process

blocks remain isolated and unaware of the other processes, allowing for easy manipulation later in the design process.

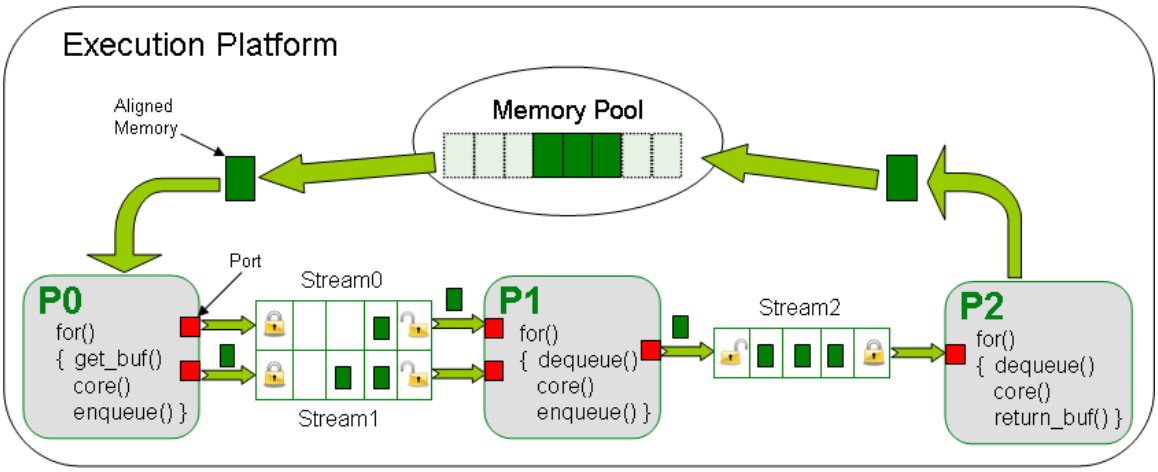


Figure 4. Execution Platform

Some of the additions to the execution platform (already shown in Figure 4) were only Linux-specific but just as necessary when providing support for the Cell, which had Fedora Core 6 installed on it. The package's Makefiles for compiling the kernel and applications provide compilation rules for ix86 machines running Microsoft Windows, a PowerPC running VXWorks, Xilinx Virtex-II FPGAs, and now include rules for the Cell running Fedora Core 6 that link with the Makefiles accompanying IBM's Cell Software Development Kit (SDK) [7][8]. This required some additional targets, variable renaming, and meticulous scripting to force the SDK Makefiles to match the file structure that would result if one was to compile with, say, Microsoft Visual Studio on an ix86 machine. The platform will compile successfully for both the GNU Compiler Collection (GCC) [3] for the Cell and IBM's XLC/C++ [11].

One of the more significant adjustments to the platform is that the kernel executes process blocks concurrently in the Linux environment. Previously, the run-time kernel only scheduled and ran the signal processing blocks in a round robin, sequential manner. Now, when it detects the new environment through __PPU__ directives, it implements concurrency by creating a POSIX thread (or, pthread) upon the first execution of each process. These threads run the signal processing blocks, which now require a simple infinite loop around execution code that is required to repeat. This proved to be a wise decision for when programs are loaded and run on SPEs. Structures called SPE contexts are created and the SPE program is loaded onto the context. Then, running this SPE context initiates the SPEs to begin executing the program. The context run function is a thread-blocking call. Therefore, this thread setup is the only way to run multiple contexts simultaneously.

Another set of necessary additions to the kernel revolved around the Cell's (more specifically, the SPE's) strict memory alignment stipulations, as mentioned earlier. Thus, all allocations of memory are now at least quad-word aligned in anticipation of transfers to a SPE's LS.

Lastly, in order for data to safely flow through the concurrent system, the communication streams between the threads are synchronized and treated as critical resources. Mutexes nicely control the access to interprocess communication streams and produce thread stalls upon denied requests, as opposed to wasteful polling loops. A denied request occurs when a thread attempts to retrieve data from an empty stream queue or if it attempts to push

data onto a full stream queue. The kernel, as before, handles the stream connections between processes and now associates the mutexes and unique process keys to the streams.

Code Generator

One part of preparing the modeled system for execution involves a code generator (the SPML WrapperGen Interpreter) that produces the wrapper code for each defined component. Based on the component design, it generates the code that will interact with the kernel in order to bring in the appointed data, allocate new memory, and prepare it for passing into the core function. It then generates clean up code and code for sending on the processed data after the core returns.

For each designed component, the naming and data typing of the variables and the name of the core function are all directly obtained from the SPML model. The core has a filename attribute associated with it. This is the file the user is responsible for producing. He or she needn't worry about the movement of the data, but simply how the data is handled once in the core function. The benefit of this is the modularity of the core. Once the core function signature matches what the code generator expects, it can perform the task however the user chooses. Algorithms can be swapped in and out as long as the signatures match. Figure 5 shows how a modeled FFT component has a core called fft_core and a FileName attribute of fft_core.c. The generated code, FFT.c, expects the fft_core to be defined elsewhere and proceeds to call it when

necessary. The user must then provide the fft_core.c file, with the fft_core function defined and matching the signature expected by the generated code.

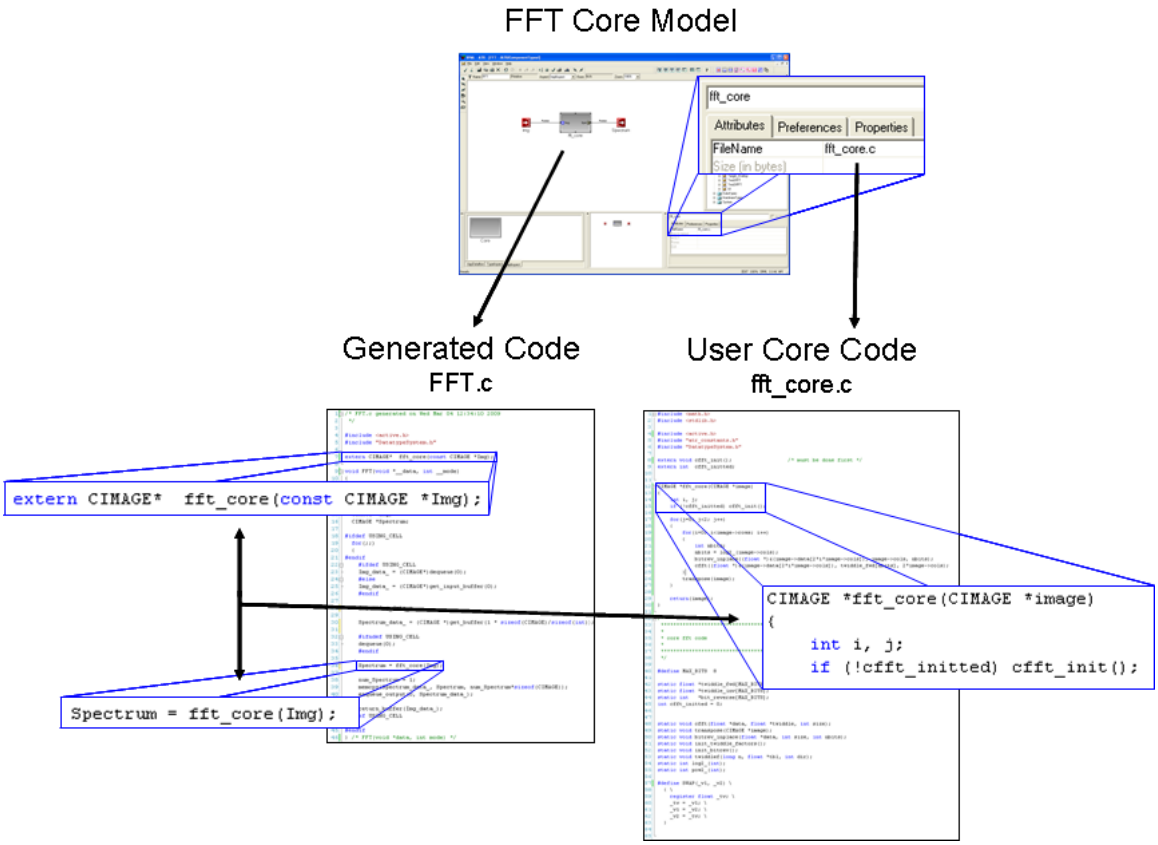


Figure 5. Managing Code

In order for the components to continually run in the concurrent environment, an infinite for-loop is added around the generated code with a directive that checks for the Cell environment. Also, within these directives, the wrapper code isn't required to check on the stream availability—it will simply hold until the kernel returns with the requested data or having pushed out the processed data.

In addition to the wrapper code, the code generator produces a header file containing all the data types that the user defined in GME. Then, each component's wrapper code includes this file in order to use the defined data types. Due to the memory alignment issues, every data type needs to be quad-word aligned in case it is transferred to an SPE. This issue mostly deals with defined structures and is fixed by keeping track of the primitive data types in the structure as it is generated. Once they are added, an array of unsigned characters (one byte each) is added to produce proper alignment. The size of the padding array is calculated by the equation:

$$pad = 16 - totalPrimByteSize\%16 \quad (1)$$

with totalPrimByteSize being the total number of bytes occupied by the primitive data types in the structure. Following the padding, any user-defined data types are added. Since the user-defined data types are properly aligned themselves when defined, the alignment will hold when they are added to other data types.

The largest and most valuable addition to the code generator revolves around generating code for running a core on one of the Cell's SPEs instead of the PPE. Currently, the user is faced with some responsibilities—selecting what components to run on an SPE and ensuring the code and data will fit in the LS—but he or she typically isn't burdened with the necessary tasks of setting up an SPE to run the process and moving the data in and out of the LS. Additionally, users have the option of using an Allocation attribute that was added to Ports in a

component in order to designate how the data should be declared on the SPE. “Dynamic” allocation generates a pointer to the data structure and line in which malloc_align() is used to acquire space for the data while selecting “Static” declares the data structure for compile-time allocation and stores the base address in the pointer. Typically, this choice should not matter; however, our example will cover an instance where these selections play an important role. After the user designs a component and decides to run it on an SPE, he or she simply adds a file name (in addition to the file name corresponding to the core function definition) to its attributes with “_spu.c” appended to the end and the code generator identifies it as an SPE component.

First, the code generator produces wrapper code that diverges from the original code. It adds code using the API from the runtime management library [9] to create a SPE context, load the core program on the SPE, and run the context. The method for providing the SPE with the necessary data involves the generation of a context_data structure that holds the effective address (EA) of all pertinent data (which correspond to the ports in the component). Then, the EA of that structure is passed to the SPE upon execution of the context. Once again, the structure must meet alignment requirements so a pad array is added to the structure. The initialization of the context_data structure and the call to run the SPE context replace the call to the core function, which will now reside in the SPE source code.

Then, the code generator creates a separate folder with “_spu” appended to it that corresponds to the core. This is where the SPE program will be built (it

is compiled separately). It generates a Makefile for compiling the program and and the "_spu.c" source file that was specified for the component in the model. The Makefile defines the program name to be the same as the folder name and it links with each object file corresponding to the file names included in the attributes of the modeled component. The source file first contains the same context_data structure that is defined in the PPE wrapper code. Then, mirroring the wrapper code, the generator declares the variables corresponding to the ports of the component and allocates properly aligned memory for them (based on the selected allocation method). It also declares the context_data structure and variables that will hold the 64-bit EAs of the data in the context_data. First, a MFC command is issued for importing the context_data structure. Then, after acquiring the individual EAs, each data element is brought in through successive MFC commands. With all data imported into the LS, the core function is invoked. When it returns, the data corresponding to the output ports is moved to main memory for transfer by the kernel on the PPE and clean up code concludes the SPE program. The only addition needed from the user is flags added to the IMPORTS variable in the Makefile that correspond to any libraries used by the core function he or she has defined. The user is still only asked to spend most of his or her resources developing the computational core, and the ability to swap core algorithms remains intact even on the SPE.

Interpreter Tool Chain

The SPP interpreter tool chain, shown in Figure 6, is a series of tools that map a SPML model to the dataflow execution platform. The first three (SPML2Desert, Desert, and Desert2SPML) are for using the DESERT [18] tool for models that have multiple potential configurations.

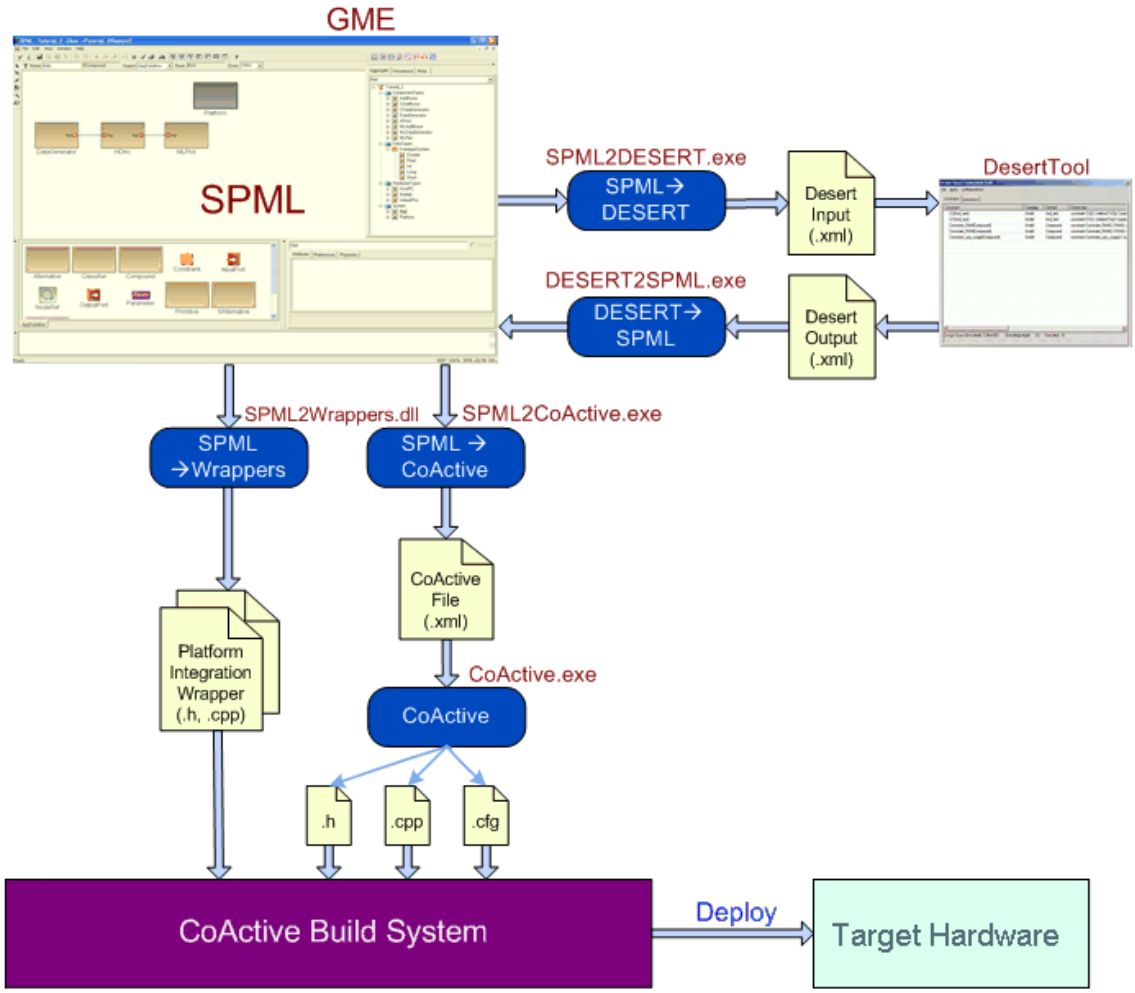


Figure 6. SPP Tool Chain

These tools allow the user to choose one configuration (typically selecting hardware platforms for the components) and then another SPML model is produced with the selected configuration only in place. This model is used for the next tool, SPML2CoActive which flattens the model and prepares an XML file for the CoActive tool. The CoActive tool is the entry point of the execution platform. It takes the flattened system and synthesizes the necessary glue and configuration artifacts such as communication maps, schedule tables, and interface specifications for the platform. The final available tool in the chain is the build tool which will use Microsoft Visual Studio to compile the generated application files on the ix86 CPU running Microsoft Windows, if it is available.

One tool-wide update was the addition of CELL as an accepted CPU type in the model. The tools simply carry this attribute through their processing until it reaches the CoActive tool. One of the components the CoActive tool produces is a files for a platform-specific system folder—a main.c source file that contains a unique definition of main() and an associated Makefile. For example, ix86 systems running Microsoft Windows would have a system folder labeled “win-ix86” that contained these files. The CoActive tool now generates for a folder named “fc6-cbe” for the Cell CPU.

Because the build tool targets the ix86 system, it actually is unneeded for an application aimed at the Cell. Proper procedure for the Cell involves transferring the generated application files to the SPP file structure on the Cell system and running `make` at the top level. The Cell compiler (XLC or GCC) will use the pre-built and generated Makefiles to build the application for the Cell.

CHAPTER V

AUTOMATIC TARGET RECOGNITION EXAMPLE

The example system used for monitoring the updated SPP is an embedded real-time Automatic Target Recognition (ATR) system as taken from [1]. This image-processing system finds and classifies the objects in the input images that belong to a set of target classes. The main processing steps involve 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 centers of regions of interest (ROIs) that are processed further and identified using a Distance Classifier Correlation Filtering (DCCF) algorithm, which increase the confidence level of the identified peaks.

The system uses images of 128×128 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 two-dimensional fast Fourier transform (2D FFT) on the image in order to correlate the images in the frequency domain. Multiple copies of the current image pass through a computational pipeline—one for each possible target class with which the image can be correlated. A maximum average correlation height (MACH) 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.

Modeling

The first step to building the application involves using the SPML to model the system: the data types, the components, the potential hardware platforms, and the dataflow of the system. First, primitive data types are defined (Float, Double, Int, Char) in preparation for application specific types (typically arrays and structures) that are based on these primitives. One key data type for the system is a CIMAGE structure composed of several primitive type variables and a data array of floats, as shown in Figure 7.

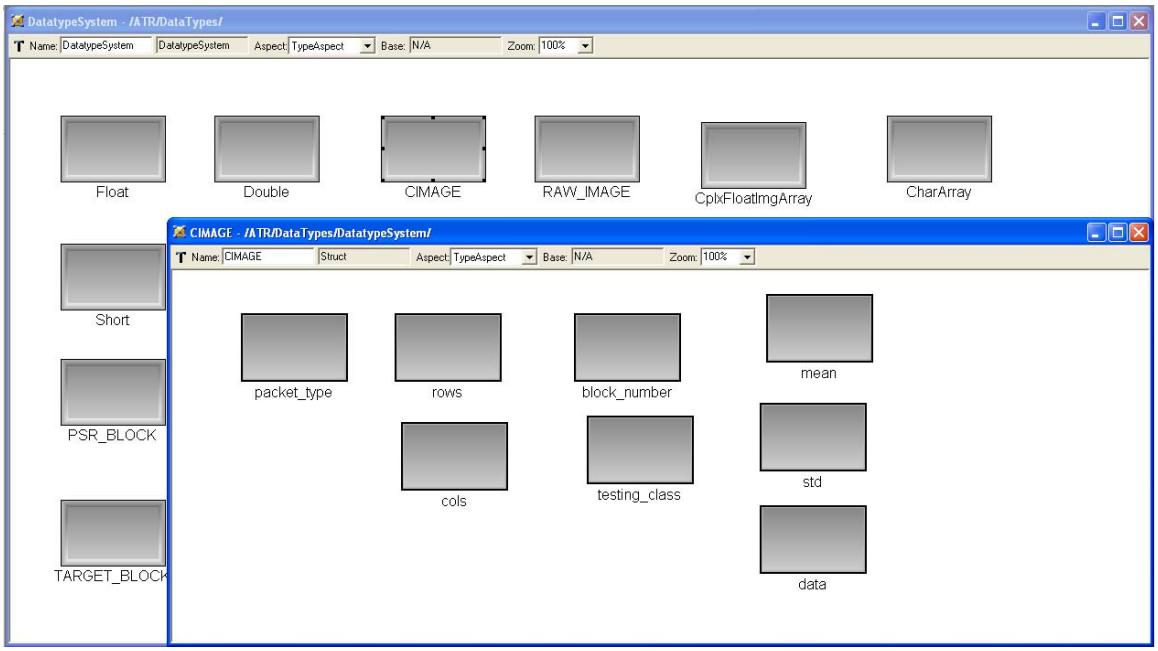


Figure 7. Defining CIMAGE Data Type

The next step involves defining all the components that the application will need to perform the required task. The components will designate tasks such as two-dimensional FFTs and IFFTs, scaling, copying, splitting, merging, correlating, etc. The following example will be for designing a component called Calculate_Distance. There are three aspects when designing a component. The AppDataflow aspect, in Figure 8, is used to define the ports of the component—the entry and exit points for the data that will be used during the execution.

Figure 8. AppDataflow Aspect for Calculate_Distance

Additionally, the NodeRef is designated for applying a hardware platform to the core (which is done later when designing the dataflow). Figure 9 shows the TypeAspect view in which the ports are associated with one of the previously defined data types. For Calculate_Distance, each port is associated with a unique data type. Finally, the ImplAspect view is where the core that represents the computational function of the component is implemented.

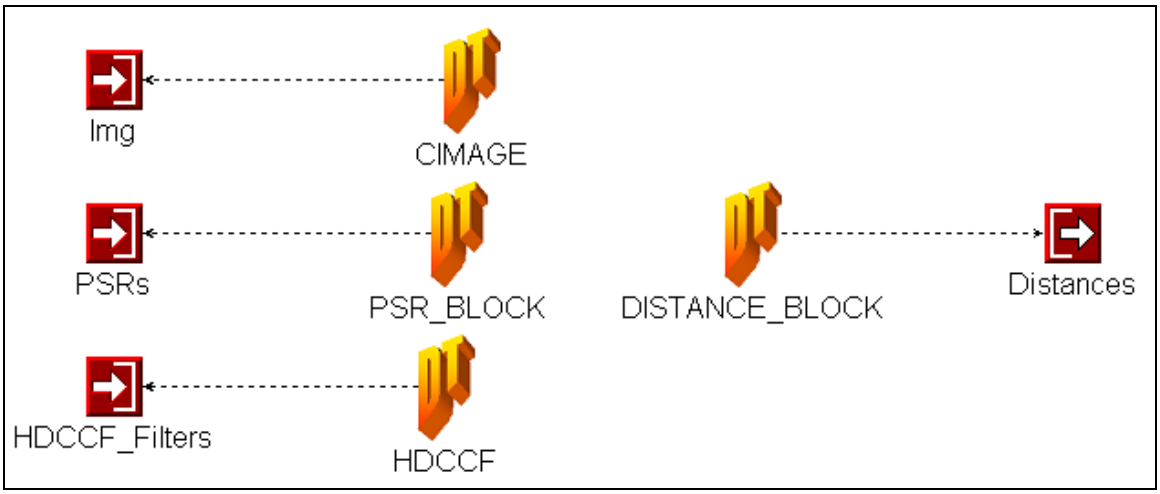


Figure 9. TypeAspect Aspect for Calculate_Distance

The Calculate_Distance core is called calc_distance_core and within it, seen in Figure 10, is each argument for the core connected to an associated data type. The expected function signature will correspond to this setup.

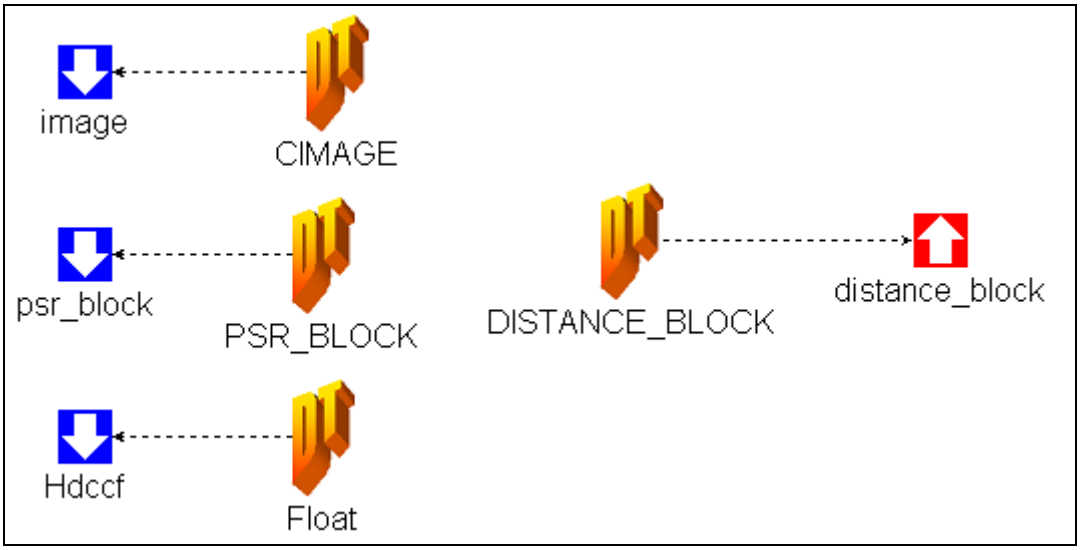


Figure 10. Design of calc_distance_core

With the core defined, Figure 11 shows how the ports of the component are connected to the arguments of the core with a pointer association. This will have pointers to the data sent to the core function.

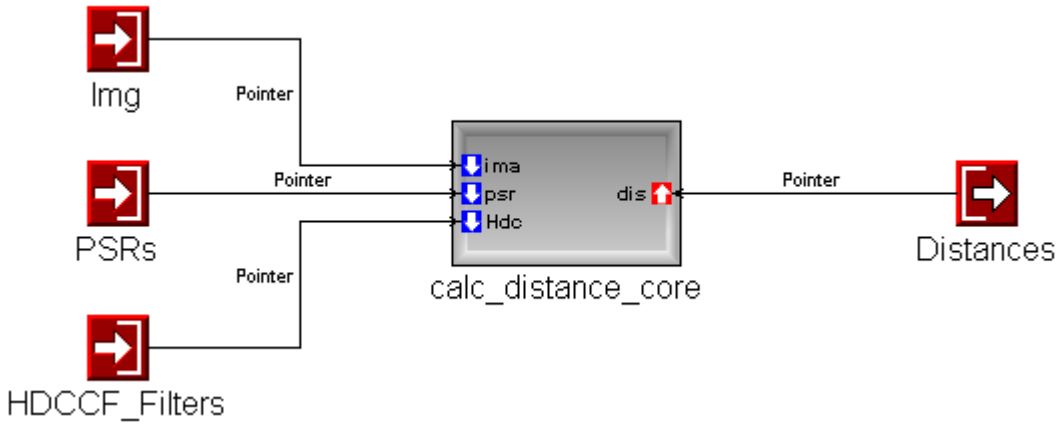


Figure 11. ImplAspect Aspect for Calculate_Distance

As mentioned earlier, the important filename attribute for the core requires the filename of the source file(s) that will contain the core function named calc_distance_core with arguments that match the designed core. For our example, this function is defined in a file name calc_distance_core.c.

After Calculate_Distance and all other primitive components are defined, SPML allows the user to build a compound component which is simply a component made up of a group of components. This is merely an organizational feature, yet rather handy when designing a large system like the ATR. The pipeline in the ATR consists of several stages for processing a copy of the image that's being processed. Having the foresight that three pipes will be needed (one for each potential target), we model a compound component called Pipe. As

shown in Figure 12, the pipe consists of a scaling block (Multiply) and an IFFT (TwoDIFFT) followed by blocks that find the mean and standard deviation of the data (Calc_Mean_Std) and calculate the PSR (Calc_PSR).

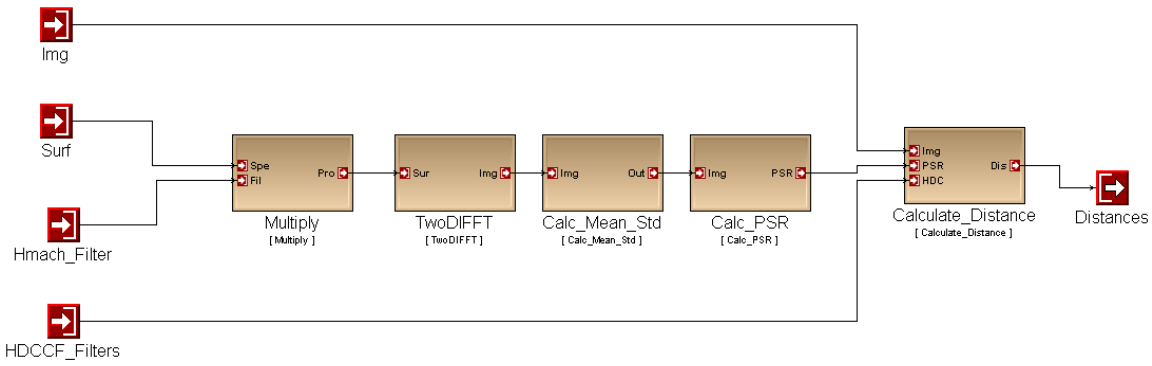


Figure 12. Design of Pipe Component for ATR

These stages correlate the incoming image with an object class from MACH filter data. The final block (Calculate_Distance) finds the distance from the other object classes based on the DCCF data [1]. After adding the necessary ports to the design, the components are connected based on the proper dataflow and the Pipe compound component is complete.

After the components needed for the ATR system are defined, before modeling the dataflow between them, the hardware type(s) the application will execute on are built. This simpler task involves choosing the type (CPU, FPGA, memory device, DSP, etc.) and filling in the necessary attributes. For CPU types, this involves selecting the type of CPU (Host, RiscPC, Cell, etc.). Then, the user is able to appoint communication ports to the hardware (serial, matlab, tcp, etc.). These are for communication between hardware types if a multi-

hardware platform is going to be used to execute the application. This is not the case for this example. Figure 13 shows two defined hardware types. HostPC is of CPUType HOST and has Matlab and Serial communication ports. Cell is of CPUType CELL with just a Serial communication port.

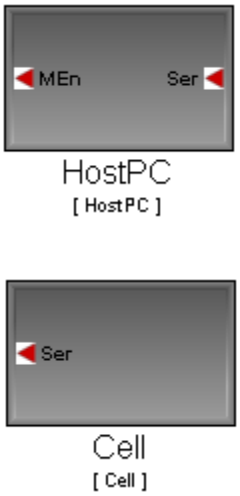


Figure 13. Defined Hardware Types

With the hardware platforms defined, the ATR components can now be combined to model the application itself. Within a system compound, copies of the predefined blocks are inserted and connected according to the dataflow of the system. Then, the hardware the application will run on is added to the model. Figure 14 shows the designed dataflow with the HostPC hardware as the selected platform for execution. Notice how three copies of the Pipe compound block are used in the model. If the grouping capability were not available, there would be several more blocks in the model, causing an unnecessary complexity and unsightliness. With this, the modeling phase for the ATR is complete.

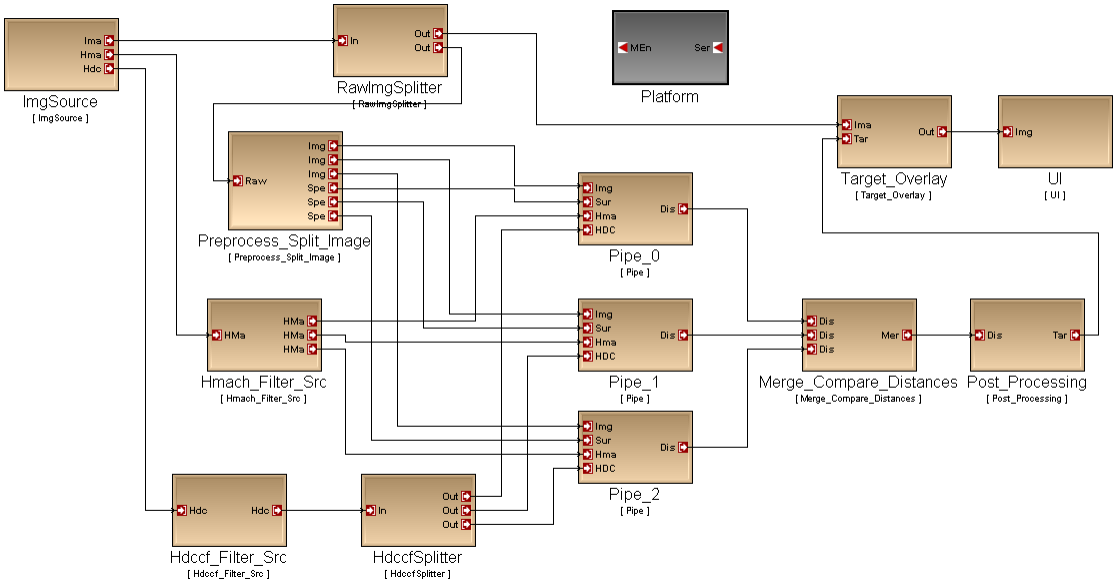


Figure 14. Model of ATR Application for HostPC

Generating Code

With the model in place, the tools in the SPP now carry the load in developing the ATR application. First, the wrapper code generator (WrapperGen), shown in Figure 15, generates the wrapper code for each defined component as previously described. Appendix E shows the code generated for the Calculate_Distance component while Appendix F contains the generated data type header file.

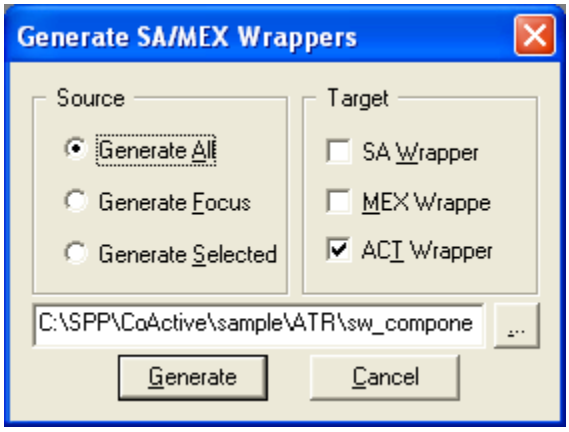


Figure 15. Wrapper Code Generator

Interpreting

Next, the SPP Tool Chain, shown in Figure 16, is invoked to interpret the model and generate the glue code and system configuration files. The first time through the HostPC is being used, which expects a windows machine running Microsoft Visual Studio. The CoActive tool expects a win-ix86 system folder and generates the appropriate code and Makefile as well as the configuration files. Appendix G contains the generated main.c file, Appendix H shows the generated system Makefile, Appendix I shows a generated process table and Appendix J shows the generated configuration file that the kernel uses to build the system at run time. Finally, the build tool is used to compile the application for the ix86.

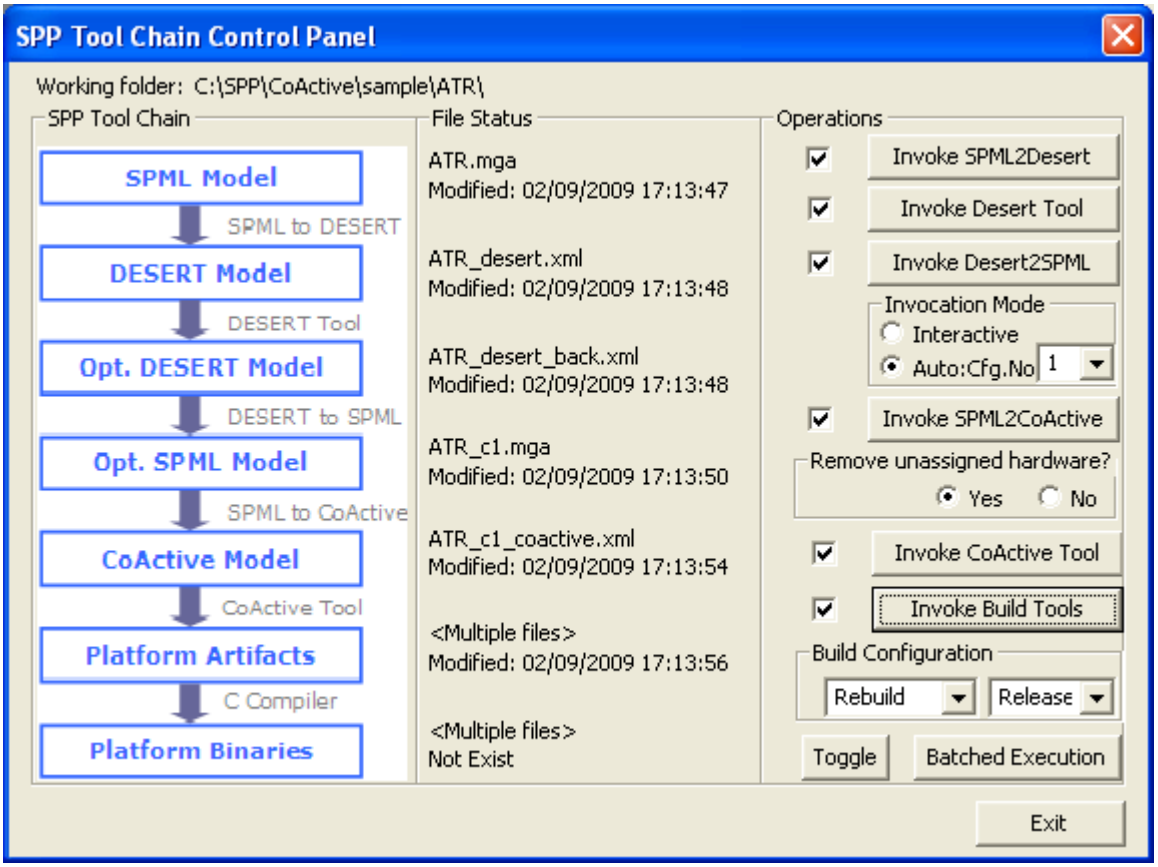


Figure 16. SPP Tool Chain Control Panel

Execution on PC

Though the output from executing the completed ATR application can be as simple as printing the coordinates of the located targets, it is much more intuitive to see the targets in the image. Thus, a GUI was added on top of the application, showing the image and the detected targets, as shown in Figure 17. Specifically, the hardware used for executing the ATR was an Intel Pentium 4 CPU running at 2.4 GHz with 1.5 GB of RAM.

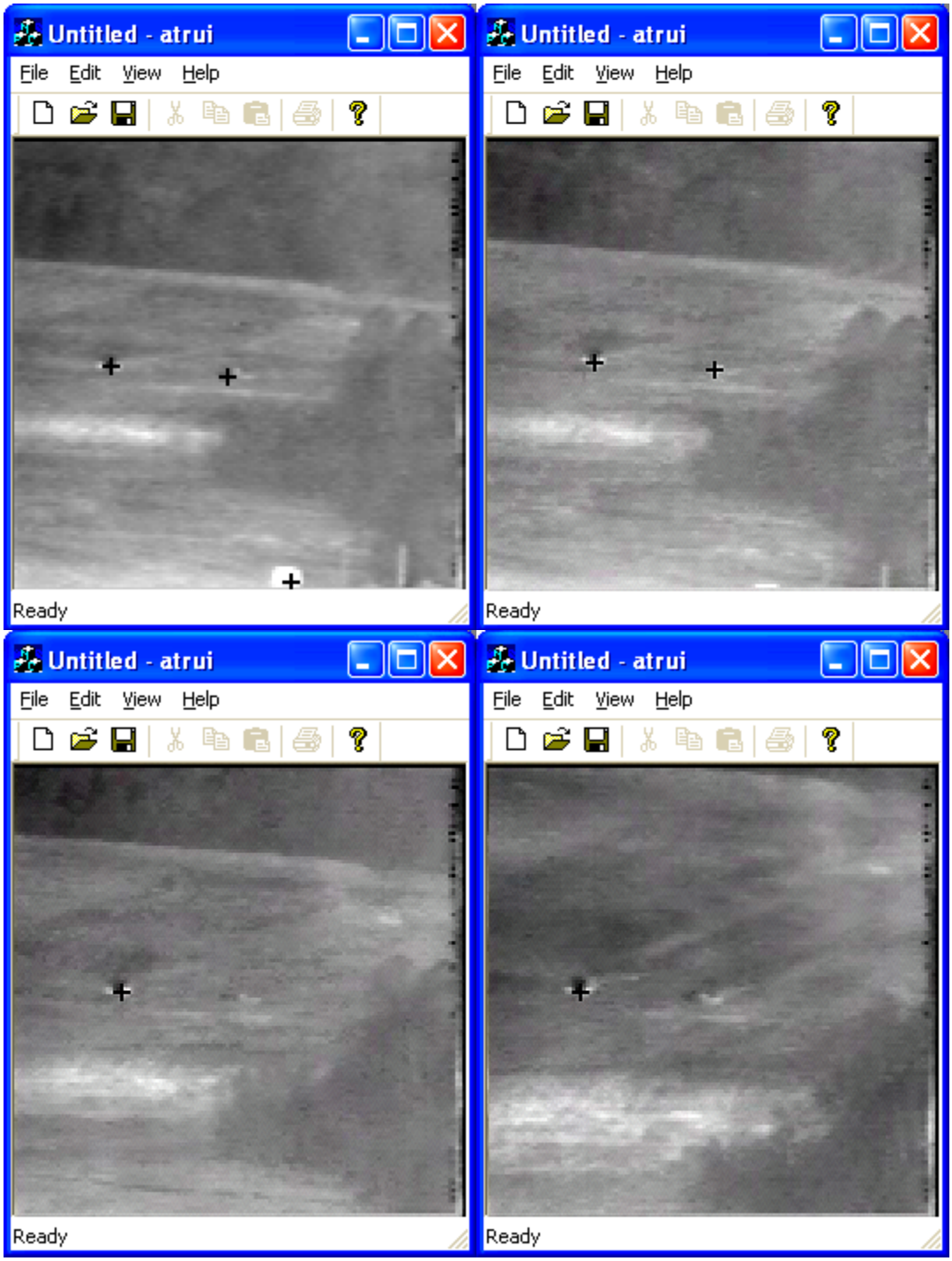


Figure 17. ATR GUI Running on PC

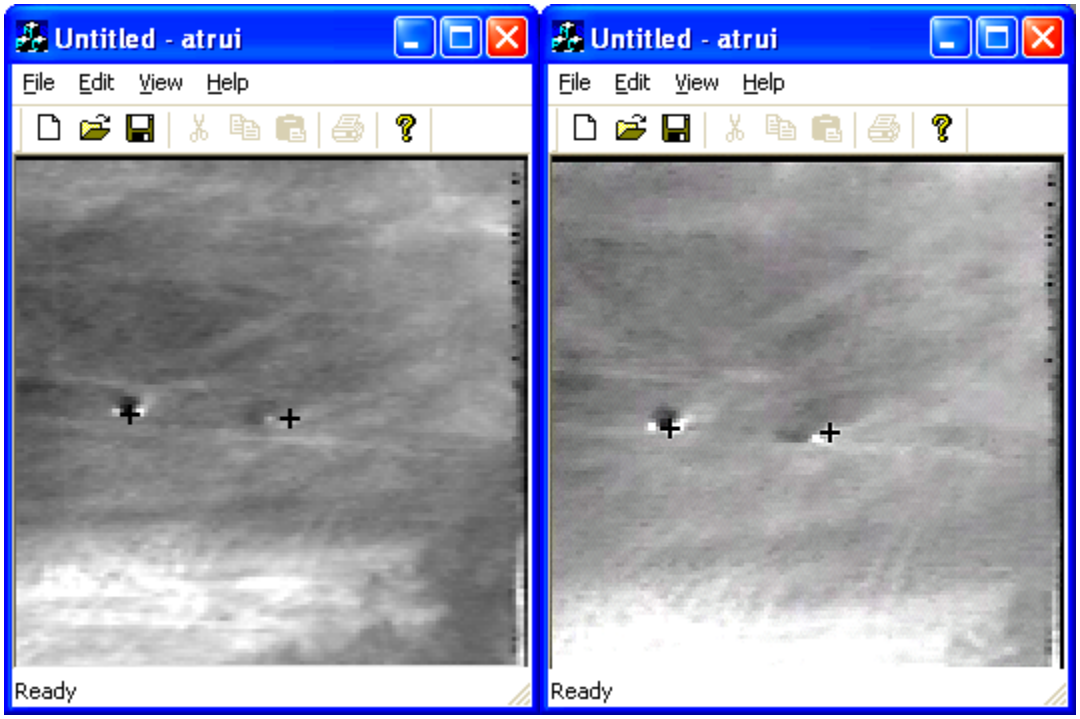


Figure 17. —cont.

Reconfigure for Cell

With a successful model, generation, and execution of the ATR on the PC, we now desire to move it to the Cell for increased performance. 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 ix86 PC, requires minimal modification even though the architectures (and even operating systems) are different.

There are just a few necessary changes in order to compile and execute the ATR on the PPE of the Cell. For any application, the user only needs to switch the HostPC hardware platform with the Cell hardware that was defined earlier and rerun the interpreter tool chain (except for the build tool). The fc6-cbe

system folder will be filled with the corresponding Makefile and definition of main(), and the configuration files will actually require no change. Appendices K and H show the Cell generated system files. Then, the system can be transferred to the Cell and compiled, and the system will execute. However, remember that the PPE is a 64-bit, big-endian core while the PC chip used is 32-bit using the little-endian convention. Therefore, there are a few pivotal changes needed to those process blocks in the ATR that deal 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 orders of the input files (the images, MACH filter data, and DCCF filter data) have to be reversed for proper reads. Again, we see the beauty of being able to easily exchange core function algorithms without disturbing the rest of the system.

Next, we make 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. Though being able to fit code and data onto an SPE's LS is the motivation, much of what is discussed are general observations of the algorithm design of the ATR. Therefore, some of the changes can be applied to the application outside the Cell as well.

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 SPE's LS along with the corresponding code. Walking through the pipeline blocks from Figure 12, the Multiply component receives an image and the MACH 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 128×128, it consumes 128×(128×2)×4 = 128 KB of memory—half an SPE's LS! What is worse is that the MACH 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 data.

The multiplied image continues through the pipeline to TwoDIFFT, Calc_Mean_Std, and Calc_PSR—none of these tasks needing much more memory for large data other than the image. The traveling image actually ends its journey in the Calc_PSR process which sends out PSRs 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 DCCF filter data. Fortunately, this filter data is only 8 KB rather than the 128 KB size of the MACH 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.

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.

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. Because 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. Quick analysis of both the execution times (on the PPE) and number of floating-point operations (FLOPs or FLOP count) for each block, shown in Table 1, provided an idea for the best division of labor.

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

Based on these values, if the first four tasks were merged into a single task, it would theoretically execute 193548 + 1179648 + 49159 + 195440 = 1617795 FLOPs in 984 + 4490 + 261 + 1326 = 7061 microseconds. The first four tasks combined are still not has hefty as the Calculate_Distance task; however, it is the best configuration for a balanced load. Thus, the pipes were redesigned to contain only two task blocks, as seen in Figure 18, and the corresponding source files for the (originally) first four tasks were glued together into one task, called Mult_TwoDIFFT_Calc_Mean_PSR. Figures 19-22 show the modeling of this new component. The input ports match those required by the original Multiply component and the output port corresponds to the output port from the Calc_PSR component. The newly defined mult_difft_calc_mean_psr_core, shown in Figure 21 and Figure 22, will actually do little more than invoke the separate cores used by the original separate components—multiply_filter_core, ifft_2d_core, calc_mean_std_core, and calc_psrs_core.

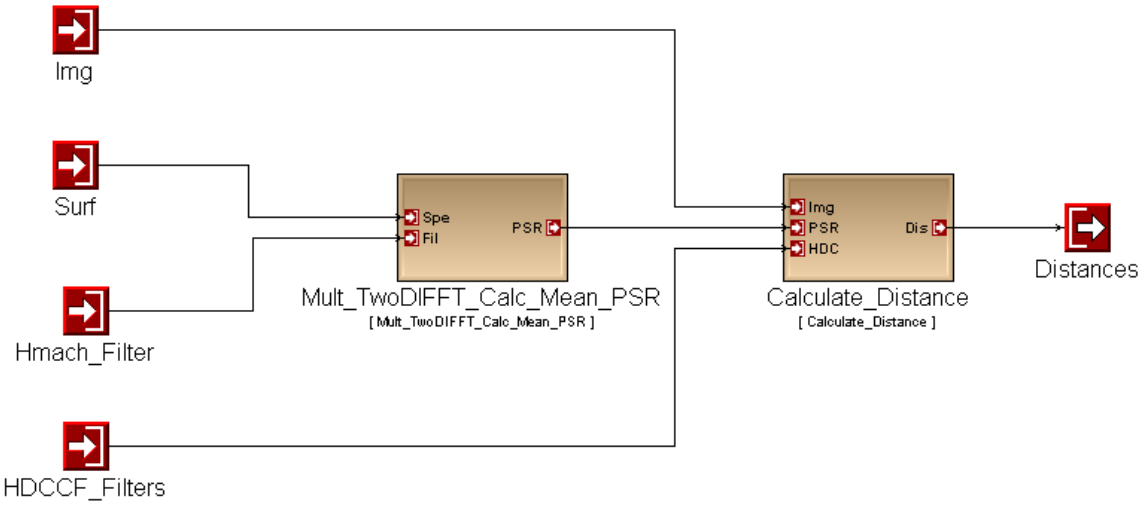


Figure 18. Structure of Merged Pipe in ATR

Figure 19. AppDataflow Aspect for Mult_TwoDIFFT_Calc_Mean_PSR

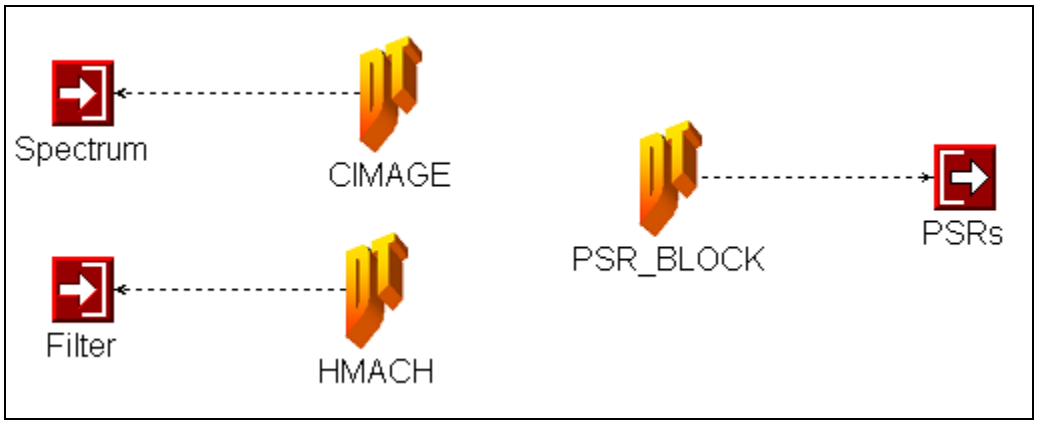


Figure 20. TypeAspect Aspect for Mult_TwoDIFFT_Calc_Mean_PSR

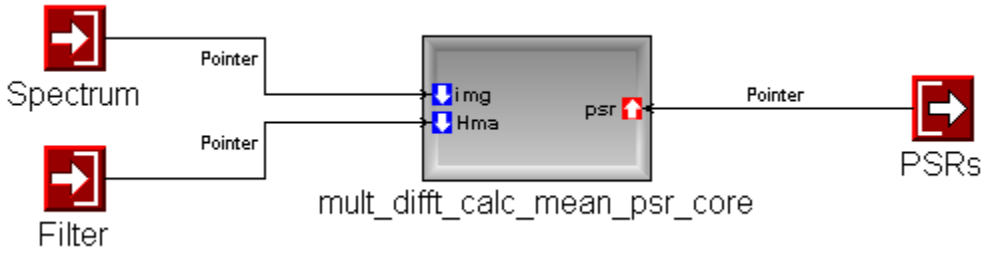


Figure 21. ImplAspect Aspect for Mult_TwoDIFFT_Calc_Mean_PSR

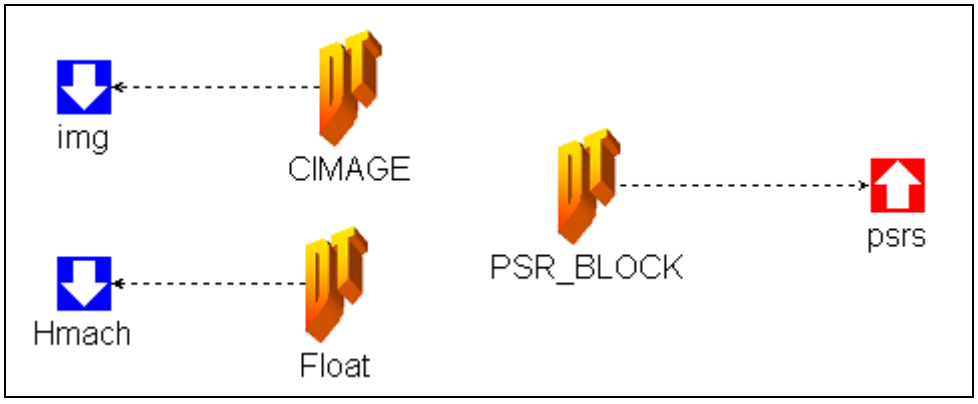


Figure 22. Design of mult_difft_calc_mean_psr_core

To support this, all the source files that correspond to the separate cores are now added to the FileName attribute for Mult_TwoDIFFT_Calc_Mean_PSR and a filename with _spu.c suffix is added to this core and calc_distance_core as well. This satisfies the task allocation requirements specified.

With sufficient task allocation, the more difficult job—resource management—remains, requiring a deep look into what data is 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 composed may be referred to interchangeably.

Recall that the large data structures existent in the first four tasks are the image and the MACH 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 in concurrence 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 MACH 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 allows us to use part of the MACH filter data on part of the image and then replace it with another part of the filter data for processing the next corresponding section of the image. It should be sufficient to store half the MACH filter (64 KB) at a time, resulting in data storage size of about 192 KB, leaving 64 KB for code (which should also be 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 32×32. The full image structure, CIMAGE, is used because it matches the necessary structure for the ROI, however, the ROI uses $1/16^{th}$ of the structure. The necessary size for an ROI structure is only 32×(32×2)×4 = 8 KB, significantly smaller. By implementing a new, separate data structure for the ROI and the intermediate ROI buffer, shown in Figure 23, the amount of data required for the Calculate_Distance is reduced to about 144 KB, leaving plenty of room on the LS for the other necessary components (i.e. code).

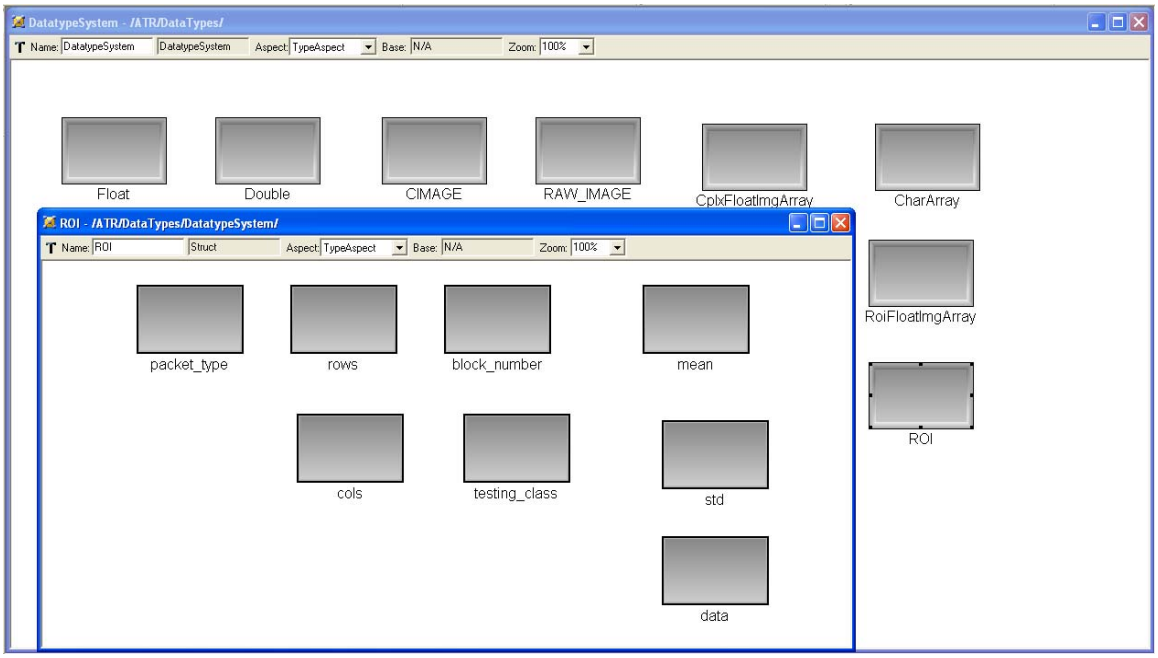


Figure 23. Defining ROI Data Type

The use of the new Allocation attribute for ports is important for this example, but unfortunately only in hindsight. Because of the compactness and thorough resourcefulness of the LS, the simple method of allocating data seemed to be the difference between data corruption and/or incorrect results and successful execution. Understanding that the location of the data in the LS differs with the allocation method as does the use of extra memory for buffering, provides justification for such behavior. Through initial trial and error, we found that a uniform allocation method—static or dynamic—would not suffice. Only a combination would produce proper results in the end. Later, deeper analysis helped reveal a better understanding of what was occurring.

For Mult_DIFFT_Calc_Mean_PSR, the quickest detected fault occurs when attempting to dynamically allocate all the major data structures. When

attempting to do this, attempting to allocate space for the filter data (after allocating for the image) always results in a NULL return, indicating a lack of space in LS. However, when declaring all structures statically, there is no complaint from the compiler about being unable to allocate LS. This time, a segmentation fault occurs when the library function ifft_1d_r2 is invoked. Observation of the stack pointer shows it in the middle of the LS at address 0xfb90 before the function call followed by a corrupted value of 0xfffff930 after the jump. More details were obtained when attempting to declare the CIMAGE dynamically but keep the other static. Using the GNU debugger and empty function calls for breakpoints, I was able to quickly isolate where the errors were occurring. However, it was discovered movement of these functions would shift the point of error slightly in the address space. This time when moving into ifft_1d_r2, the stack pointer jumps from address 0x2fbb0 to 0x1f950. Unfortunately, the data intensive library function has pointers that traverse the image and eventually overlap with this address and corrupt the stack, causing the application to fail. When the filter structure is dynamically allocated while the image and PSRs structures are static, the image is allocated starting at address 0x1fb90 while the stack pointer sits at 0xf930 in the library function. Keeping the image below the stack seems to be the solution and Mult_DIFFT_Calc_Mean_PSR executes correctly.

Similarly, for Calculate_Distance, there were arrangements that caused the program to wrongfully exit when attempting to enter a function or even just a for-loop. Declaring all structures statically cause it to quit when attempting to

enter a for-loop when adding background to the extracted ROIs. Declaring all structures dynamically managed to allow the program to execute, however the distance values are incorrect and values in the image are corrupted. Incorrect mixed combinations alternate between early exits and producing incorrect distance values. The correct combination found for Calculate_Distance is dynamically allocating the image and PSRs, HDCCF_Filters, and Distances are static structures. This is viewed as a testament to how little space remains on the LS, showing efficient usage of the SPEs. With the proper allocation methods chosen, this potential problem is avoided.

With all the updates in place, WrapperGen will generate the new Wrapper code and the corresponding spu files. Appendix L shows the alternate wrapper code for Calculate_Distance while Appendix M contains the corresponding code that is generated for the SPE itself, and Appendix N shows the generated Makefile for the SPE. The PPE acquires passed in data but then sets up the context_data structure, sets up the SPE context, runs the context with the structure passed in, and waits for the context to return. It then finishes by passing forward the processed data. The SPE algorithm involves moving in the context_data structure, moving in the data referenced in the structure, running the core function, and finally moving the processed data back to main memory before returning.

Unfortunately, for the ATR, some minimal user intervention is necessary. Because of the necessary changes to the Mult_TwoDIFFT_Calc_Mean_PSR algorithm described above, the generated SPE code needs to be changed to

reflect this. As shown in Appendix O, the changes (in bold) involve allocating and using only half the MACH filter data at a time. Though, there should only be one core call to mult_difft_calc_mean_psr_core, it first calls the multiply_filter_core function and uses the first half of the MACH filter data. Then, it brings in the second half over top of the first half and calls the proper core, which will call the multiply_filter_core function and expect the second half of the filter data. The necessary changes are minimal and the user does not spend as much as time as he or she could writing this code from scratch.

With the task allocation and resource management resolved, we now have the updated ATR model with the Cell as the targeted hardware, shown in Figure 24, and can rerun the interpretive tools to generate the proper system files for the application.

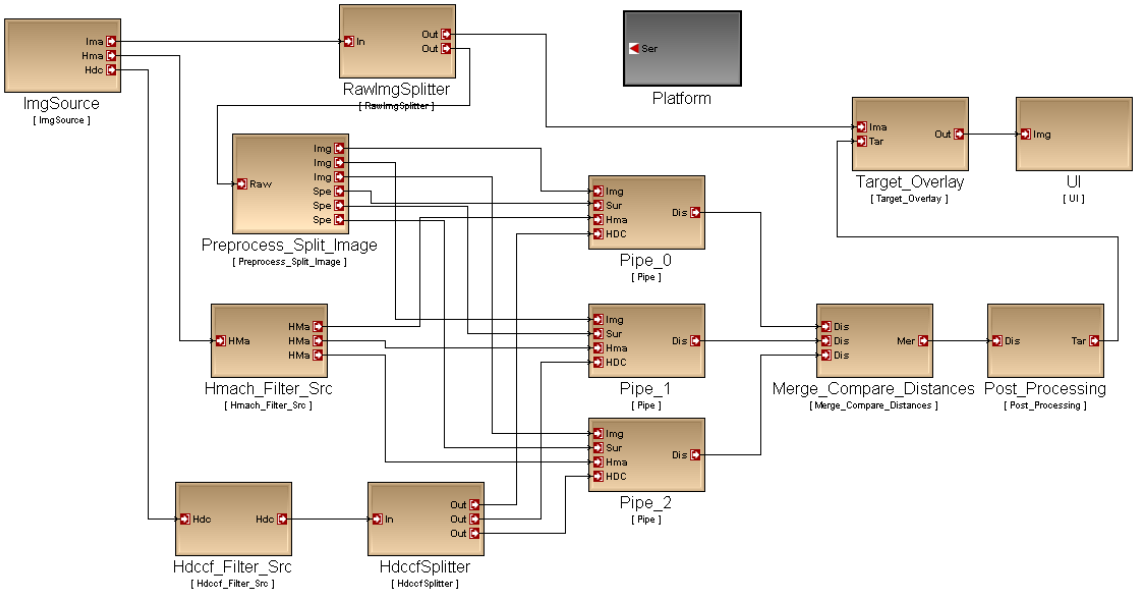


Figure 24. Model of ATR Application for Cell

Replacing Cores for the Cell and Code Optimizations

In addition to the new cores resulting from the merged components, we now take advantage of the interchangeability of core functions in order to develop cores that take advantage of the Cell's capabilities. The original algorithms for fft_2d_core and ifft_2d_core are replaced by methods that take advantage of the FFT library provided by IBM's Cell SDK 2.1 [7]. In addition, we replace algorithms for multiply_filter_core, calc_psr_core, and calc_distance_core with techniques that utilize the vector support of the Cell—operating on four floating-point values at a time. Some data shuffling and a few other minor data manipulation instructions are necessary to meet data arrangement requirements when using these vector instructions, but the exceptional performance increase remains.

Finally, with all necessary code having been generated, the user is perfectly capable of making whatever modifications he or she desires (similar to the necessary changes to Mult_TwoDIFFT_Calc_Mean_PSR mentioned earlier), typically in order to further increase performance. For the ATR, we observed a redundancy in repetitive acquisition of static filter data. By storing these structures locally with the processes (Multiply and Calculate_Distance), it reduces delays due to waiting for the data and unnecessary copying.

Execution on Cell

After a successful compilation on the Cell, we execute the ATR with a GUI overlay similar to the PC version, shown in Figure 25.

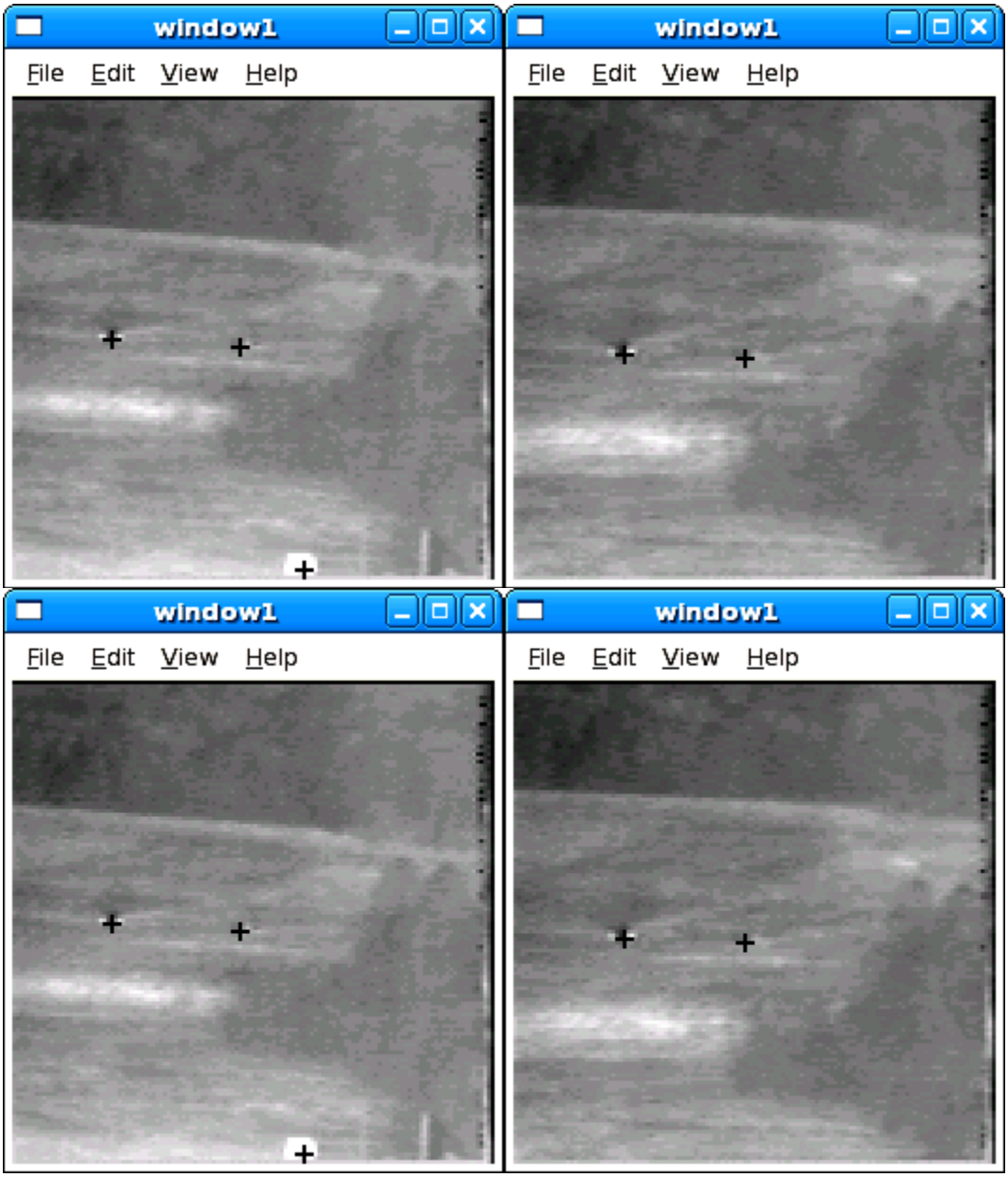


Figure 25. ATR GUI running on Cell

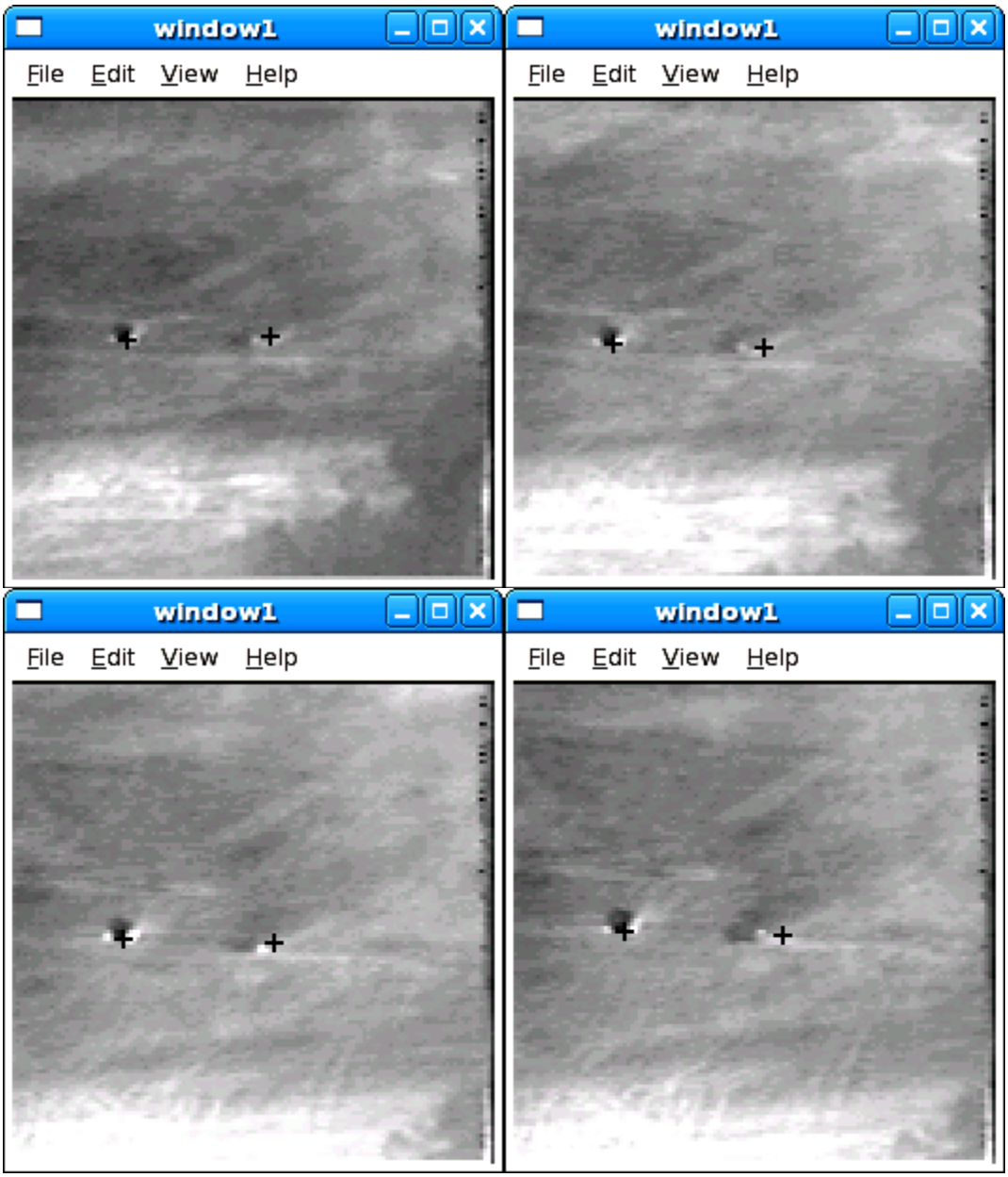


Figure 25.—cont.

CHAPTER VI

RESULTS

We used ATR applications that mirror those presented with the exception that the GUI layer is removed to measure the performance of our developed system. 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 averaging ten averages of run time measurements of one thousand processed images. Knowing the breakdown of FLOPs for each block per image, shown in Table 2, and the size of an image (128 KB) allows for computation of the various performance parameters.

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
HdccfSplitter	0
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Mult_TwoDIFFT_Calc_Mean_PSR	1617795
Calc_Distance	1948448
Calc_Distance	1948448
Calc_Distance	1948448
Merge_Compare_Distances	0
Post_Processing	47
Target_Overlay	0
UI	0
TOTAL FLOP COUNT	**11845656**

Table 3 contains those performance results for the ATR running on the Intel Pentium 4 and the two ATR applications that take advantage of the Cell—the second resulting from the manual optimizations.

Table 3. Performance of ATR

	ATR on PC	ATR on Cell	ATR on Cell Opt
Framerate (frame/s)	35.71	219.45	247.77
Throughput (Mbit/s)	37.45	230.11	259.81
MFLOPS	423.06	2599.55	2935.02

As is shown, using the Cell increased the performance of the ATR system dramatically. The introduction of physical parallel computing and operations on multiple data values at a time prove useful in obtaining a faster signal processing application with little additional work.

Even with these indisputable results, the following data helps to breakdown the source, reasoning, and limitations for the increased performance. Table 4 shows the major sections—both data movement and data crunching stages—of the Mult_TwoDIFFT_Calc_Mean_PSR algorithm running on the SPEs with their average completion times, corresponding percentage of time spent in that step, and MFLOPS achieved in each computational stage, along with the totals for the entire block. Because there is no cache on the SPEs, the average data crunching run times are consistent to within ±0.01 microseconds. Note that the FFT libraries from the SDK help produce nearly four GFLOPS, and it is quite apparent the calculate_psrs_core function, though using vector instructions when possible, needs some deeper algorithm alterations to better utilize the Cell.

Table 4. Breadown of Mult_TwoDIFFT_Calc_Mean_PSR on SPEs

SPE Section	Avg Time (usec)	Section %	MFLOPS
DMA In	56.49	2.84%	
multiply_filter_core	59.49	2.99%	1626.73
DMA In	11.25	0.56%	
multiply_filter_core	59.75	3.00%	1619.65
ifft_2d_core	310.61	15.60%	3797.84
calc_mean_std_core	15.93	0.80%	3085.94
calculate_psrs_core	947.86	47.59%	206.19
DMA Out	12.94	0.65%	
TOTAL	**1474.32**		**1097.32**

We should also take the time to point out some differences between the compilers (GCC and XLC). Table 5 shows the runtimes for Mult_TwoDIFFT_Calc_Mean_PSR resulting from being compiled by each tool, and Table 6 shows the performance for Calculate Distance.

Table 5. Mult_TwoDIFFT_Calc_Mean_PSR Run Times: XLC vs GCC

SPE Section	XLC Time (usec)	GCC Time (usec)
DMA In	56.49	56.49
multiply_filter_core	59.49	26.97
DMA In	11.25	11.25
multiply_filter_core	59.75	27.29
ifft_2d_core	310.61	226.98
calc_mean_std_core	15.93	53.58
calculate_psrs_core	947.86	947.86
DMA Out	12.94	12.94
TOTAL	**1474.32**	**1363.36**

Table 6. Breakdown of Calculate_Distance on SPEs

Context Section	GCC Time (usec)	XLC Time (usec)	XLC Section %	XLC MFLOPS
DMA In	55.11	55.11	6.84%	
calc_distance_core	882.43	749.96	93.12%	2598.07
DMA Out	0.28	0.28	0.03%	
TOTAL	**937.82**	**805.35**		**2419.38**

Looking at the tables, there are some definite differences in compilations. This application (and other simpler applications) has shown that the compilers differ at times when it comes to optimizations in areas such auto-vectorization, branch-hinting, and good instruction interleaving to take advantage of the dual-issue pipeline on the SPE. Though we see Mult_TwoDIFFT_Calc_Mean_PSR running faster with GCC, we found XLC ultimately produced better results for the entire application.

One final area worth exploring is the timeline—the flow of an image. Figure 26 shows the breakdown of the application into its process blocks. The block size corresponds to their average run times. The parallel processing is evident however, there appears to be a lot of time in which most of the threads are waiting. Remember, that the interprocess communication streams only have a depth of three. If one process has filled its outgoing stream queue, it must wait for the process at the receiving end to dequeue from the stream. Therefore, as Figure 26 is indicating, TwoDFFT running on the PPE is the heaviest process and is the runtime factor. Notice it does not wait in the timeline because all other processes are waiting on it.

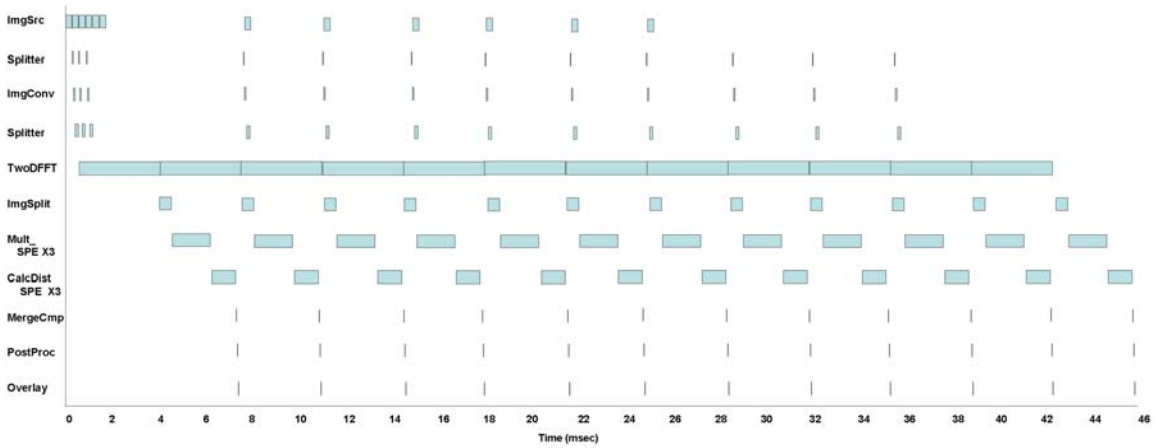


Figure 26. ATR Timeline

Furthermore, the runtime of TwoDFFT is equivalent to the spacing between runs by the Overlay process (one of the final stages for an image). Even though it uses the Cell SDK's FFT library, it would be beneficial to further improve its core or have access to another SPE. We can conclude that the most time-consuming process block's run time will closely match the average image-computation time. One final thing to remember is that balance is a crucial element. The better balanced the run times of all processes, the less waiting they will do.

CHAPTER VII

FUTURE WORK

Even while achieving excellent performance increase, there are further implementations that can possibly squeeze some additional performance as well as shift some of the remaining manual work into the automated group.

Further increased performance will most likely come from optimized core functions and, as mentioned in [6], double buffering the DMA operations on the SPEs. 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. This would be similar to the manual division of the MACH filter data for Mult_TwoDIFFT_Calc_Mean_PSR, except the segments would perhaps be smaller, allowing for mult-buffering. 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 27 shows a comparison of general operations with and without double buffering. However, this depends on the core functions. Though we could generate the code to outline such buffering, the user would be obligated to modify the core function to support smaller segment computations rather than a lump set.

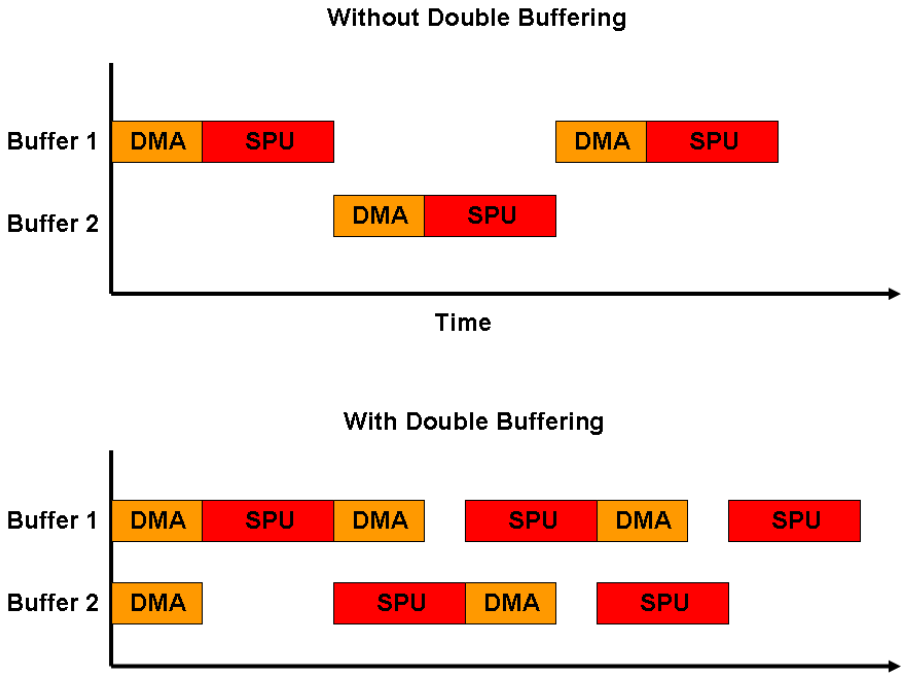


Figure 27. Benefits of Double Buffering

Further improvements for core functions involve continuing to develop Cell-specific algorithm: using SIMD instructions and unrolling loops [6]. Loop unrolling provides more instructions for the SPU, allowing for more instruction interleaving and helping to prevent any stalls that may occur due to dependencies.

Currently, work is being done to increase the automation reach of the SPP tools for the Cell. By providing some additional knowledge about each component to the model—code size and relative execution time—some additional tools will be able to analyze the modeled system and develop a schedule for running cores on SPEs. This will help replace the user's responsibility to choose which cores to run on an SPE. Furthermore, the scheduler will open the possibility to have more SPE-bound cores than available cores and/or leasing more than one SPE to one component's core. The results

of this new analysis will be reflected in the wrapper code generated and, if necessary, in some of the generated system files.

One small potential change is moving to a dynamic connection stream queue length. In the history of the SPP kernel, the queue has grown from two to three deep, solely based on a recognizable performance increase. Analysis of this situation using the ATR came to the discovery that the proper queue length for minimal wait times appears to be the lesser of the following: the number of streams between the most distant blocks or the time of one complete flow divided by the longest block runtime, rounded up. Revisiting a timeline, in Figure 28, one can see a gap between every two TwoDFFT blocks. Using the reasoning just stated, since a complete image flow takes about 7.8 milliseconds with TwoDFFT taking about 3.5 milliseconds, then 7.8 ÷ 3.5 = 2.23 which is rounded up to 3.

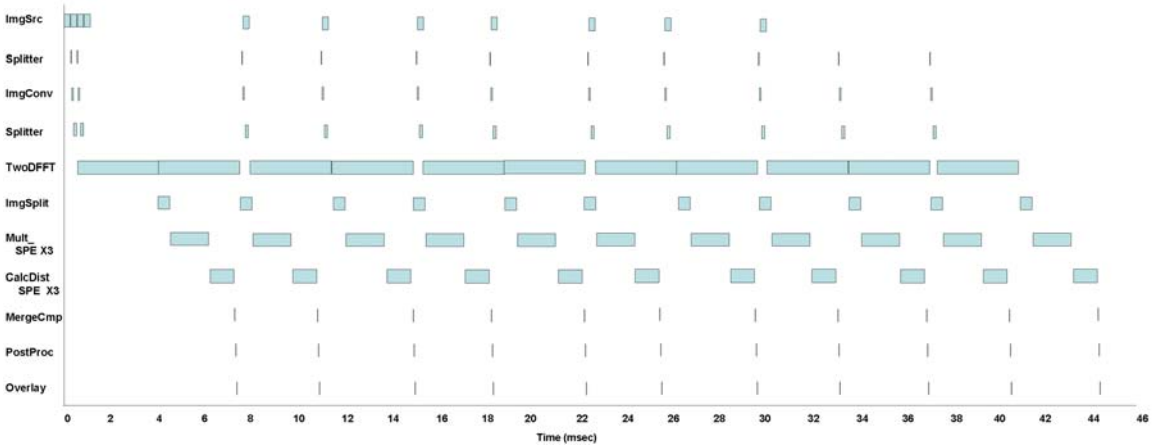


Figure 28. ATR Timeline with Stream Depth of 2

Figure 29 shows the behavior of the ATR with a stream queue size of three. Now, there is no waiting by the heaviest component and the only issue is balance

between components. Adding this type of analysis to the SPP tools could allow for the queue size to be adjusted appropriately for the modeled applications, providing optimum efficiency without waiting space (i.e. if the queue was bigger than it needed to be).

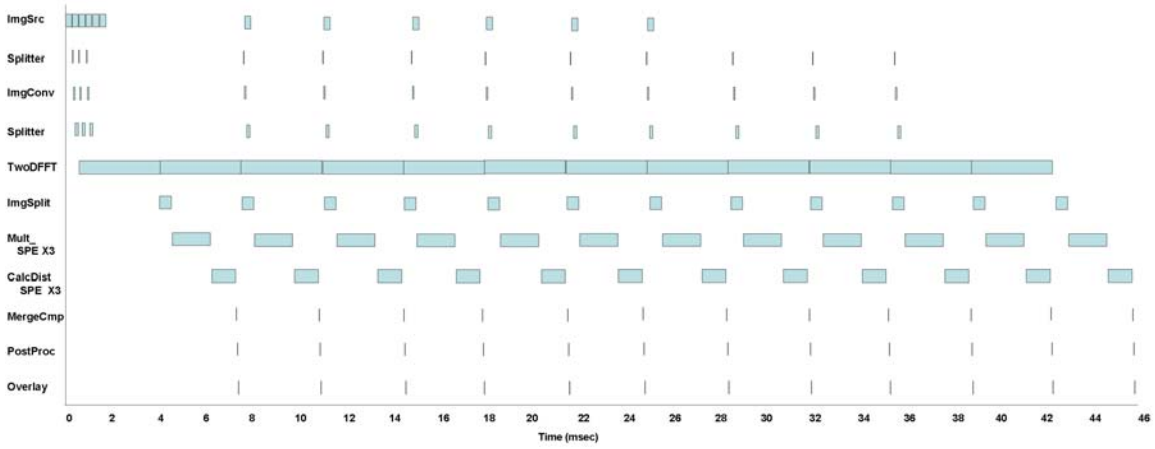


Figure 29. ATR Timeline with Stream Depth of 3

CHAPTER VIII

CONCLUSION

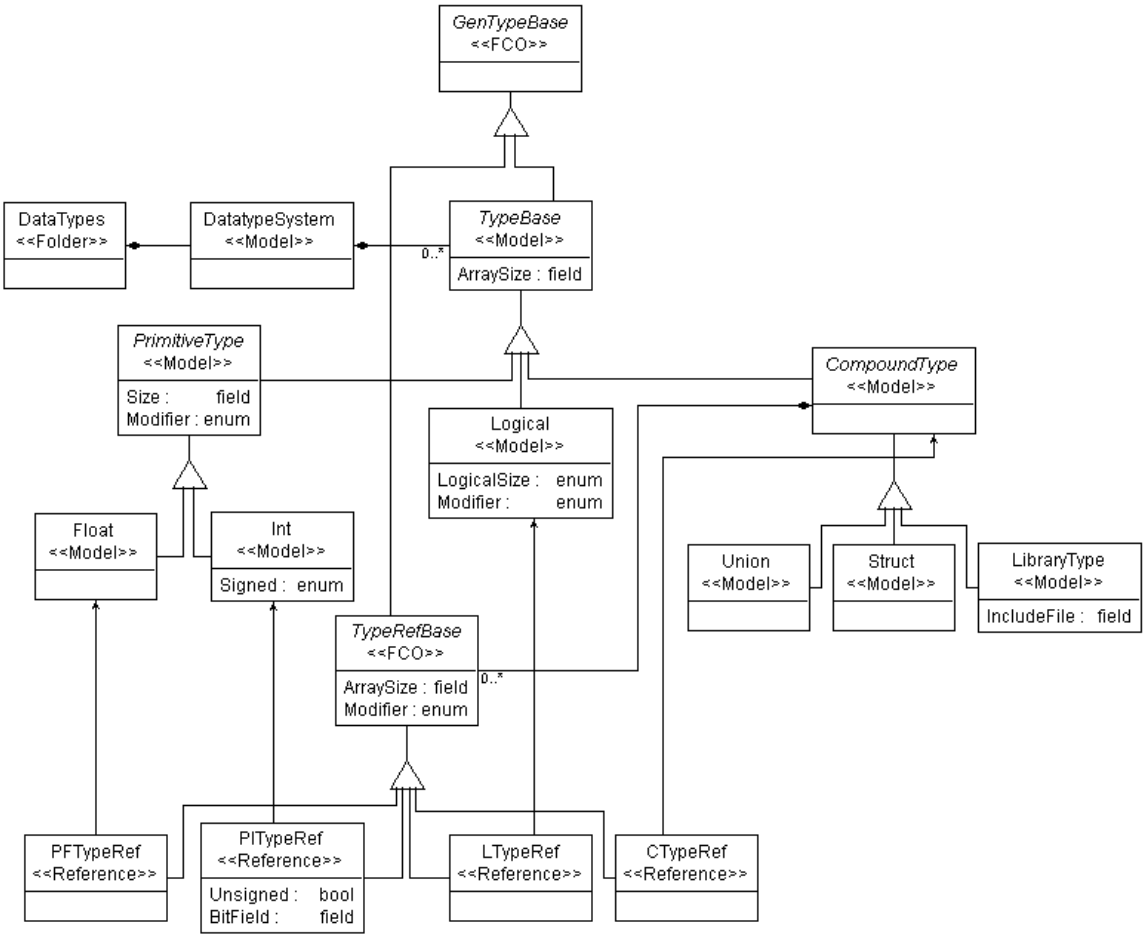
The SPP continues to be a powerful tool for modeling these complex systems and managing the runtime environment, reducing the necessary man power and time to produce a functional product. However, we can conclude that a balance between automation and human intervention will bring about optimum performance. Automation works best to create the foundation for any application while late stage manual modifications unique to the current application will unveil performance increases that are too specific for automation tools to produce. The big challenge of delaying that point of necessary human intervention remains. Due to the complexity of the Cell, the further development of automation tools must have intimate knowledge of the Cell. Perhaps instead of modifying the current stages of SPP, additional phases should be added to the tool suite when the Cell is present. The need for ambiguity would no longer be needed in order to support other architectures—the tools will be able to focus solely on fitting the system onto the Cell. These tools might involve as much as a new DSML that separates the cores while still focusing on the work and data movement. This could allow for direct movement from one SPEs LS to another and eliminate the middle stage of returning to main memory. The associated tools (at least one of which to be a scheduler like the one in development) would strive to produce a balanced workload and utilize computational cores that have

been developed specifically for the Cell. Even before this, further development of more Cell-specific files and cores that need only be produced once for the tool suite could help to reduce the necessary manual alterations.

These results suggest that updating the SPP to support the Cell processor has been a worthwhile success in providing a new platform for thoroughly increased performance with minimal extra work or knowledge on behalf of the user. It has helped to continue the progression of modeling complex embedded applications and deploying them on new, multi-core architectures. This is an important step as multi-core architectures continue to grow in popularity and complexity and the desirable systems maintain their complexity as well. Finally, we can conclude that using the SPP to develop other applications for the Cell, its SPEs, and associated libraries will present performance improvements similar to what has been presented.

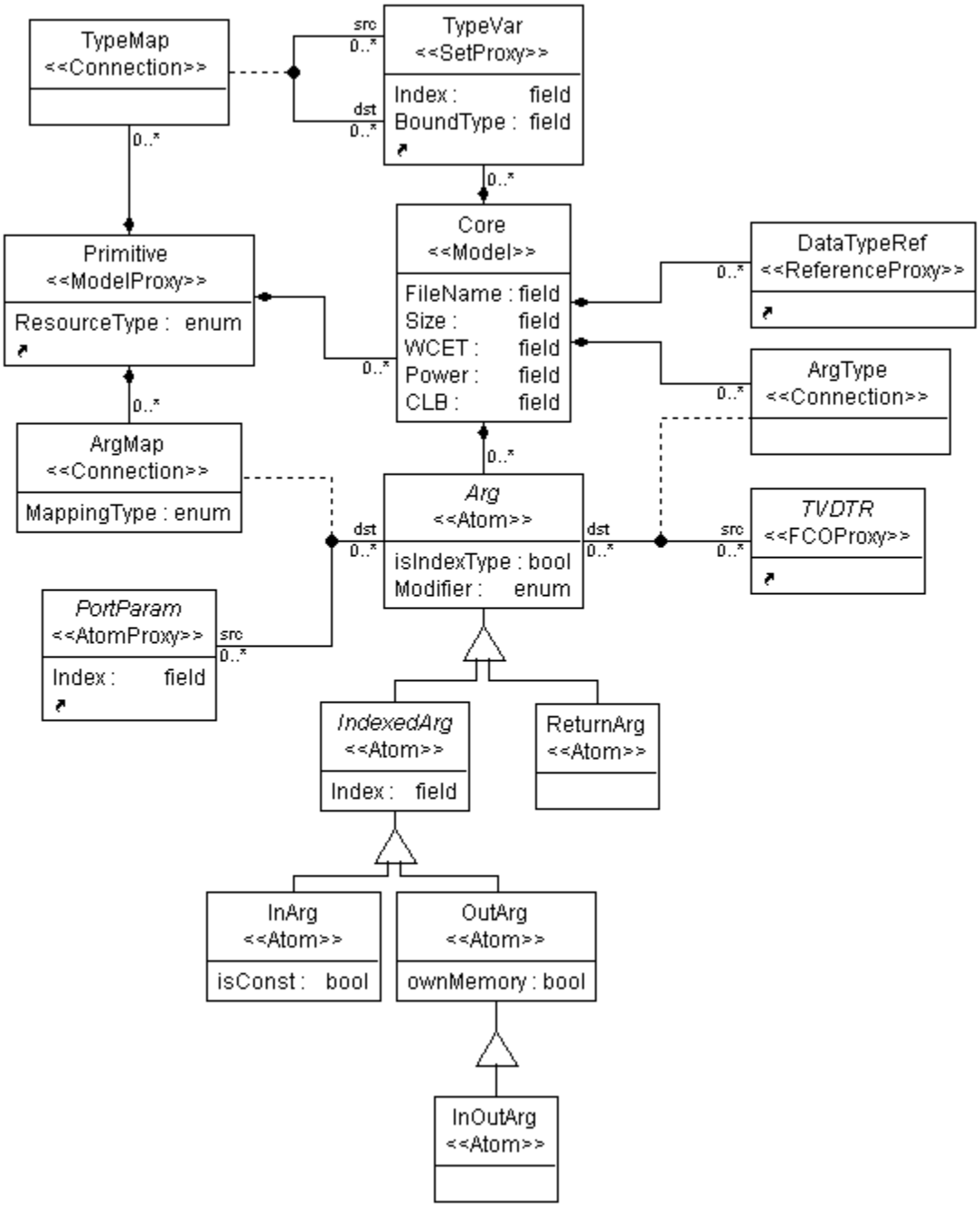
APPENDIX A

META-MODEL OF SPML DATATYPING



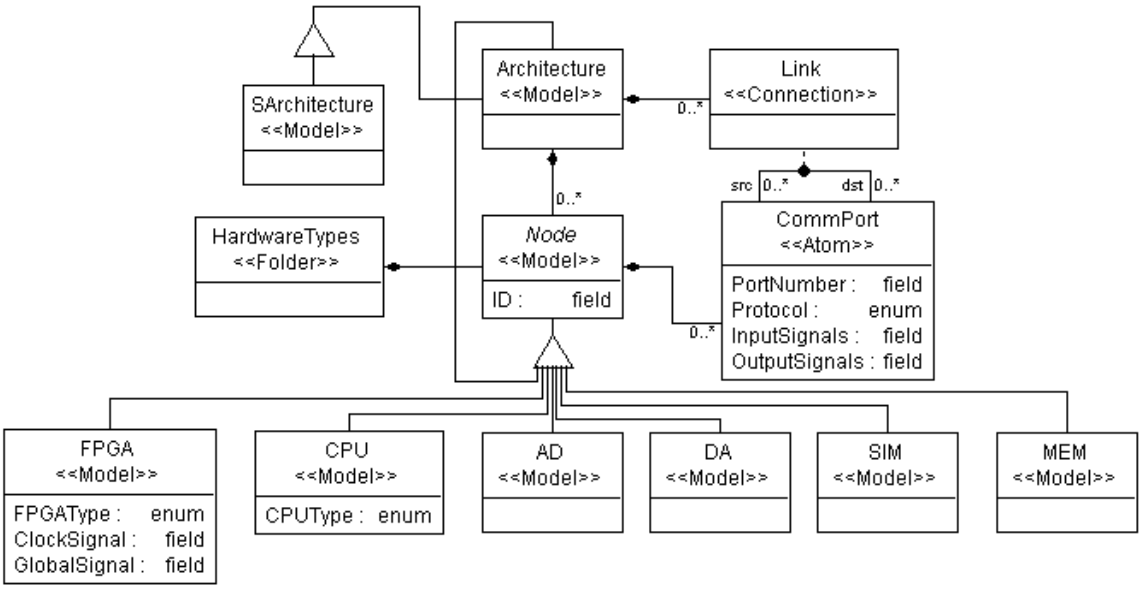
APPENDIX B

META-MODEL OF SPML CORE



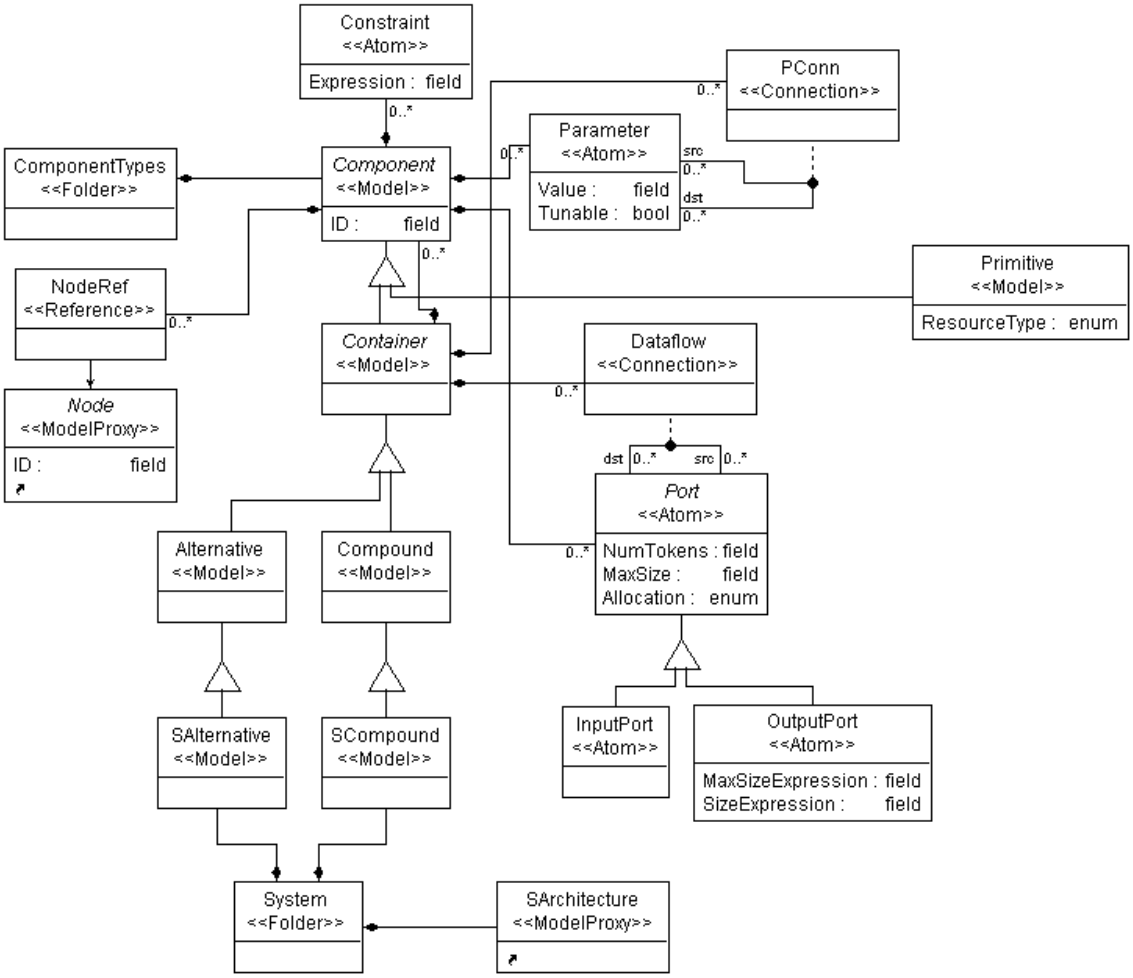
APPENDIX C

META-MODEL OF SPML ARCHITECTURE



APPENDIX D

META-MODEL OF SPML APPLICATION DATAFLOW



APPENDIX E

GENERATED WRAPPER CODE FOR CALCULATE_DISTANCE

```
/* AWCalculate_Distance.c generated on Wed Feb 18 11:33:50 2009
 */

#include <active.h>
#include "DatatypeSystem.h"

extern void  calc_distance_core(DISTANCE_BLOCK *distance_block, HDCCF
*Hdccf, PSR_BLOCK *psr_block, CIMAGE *image);

void Calculate_Distance(void *__data, int __mode)
{
  CIMAGE *Img_data_;
  PSR_BLOCK *PSRs_data_;
  HDCCF *HDCCF_Filters_data_;
  DISTANCE_BLOCK *Distances_data_;
  int num_Distances;

  CIMAGE *image;
  PSR_BLOCK *psr_block;
  HDCCF *Hdccf;
  DISTANCE_BLOCK *distance_block;

#ifdef USING_CELL
  for(;;)
  {
#endif
    #ifdef USING_CELL
    Img_data_ = (CIMAGE*)dequeue(0);
    #else
    Img_data_ = (CIMAGE*)get_input_buffer(0);
    #endif
    #ifdef USING_CELL
    PSRs_data_ = (PSR_BLOCK*)dequeue(1);
    #else
    PSRs_data_ = (PSR_BLOCK*)get_input_buffer(1);
    #endif
    #ifdef USING_CELL
    HDCCF_Filters_data_ = (HDCCF*)dequeue(2);
    #else
    HDCCF_Filters_data_ = (HDCCF*)get_input_buffer(2);
    #endif

    #ifdef USING_CELL
    if( !(Img_data_ && PSRs_data_ && HDCCF_Filters_data_) )
    {
      /* no data to execute or no room to put results */
      continue;
```

```
    }
    #else
    if( !(Img_data_ && PSRs_data_ && HDCCF_Filters_data_ &&
output_slot_available(0) ) )
    {
      /* no data to execute or no room to put results */
      return;
    }
    #endif

    image = Img_data_;
    psr_block = PSRs_data_;
    Hdccf = HDCCF_Filters_data_;

    Distances_data_ = (DISTANCE_BLOCK *)get_buffer(1 *
sizeof(DISTANCE_BLOCK)/sizeof(int));

    #ifdef USING_CELL
    if( !Distances_data_ )
    {
      /* bad output pointer */
      active_error(__LINE__, ACT_MEMORY_OUTOFMEMORY);
      continue;
    }
    #else
    if( !Distances_data_ )
    {
      /* bad output pointer */
      active_error(__LINE__, ACT_MEMORY_OUTOFMEMORY);
      return;
    }
    #endif

    distance_block = Distances_data_;

    #ifndef USING_CELL
    dequeue(0);
    #endif
    #ifndef USING_CELL
    dequeue(1);
    #endif
    #ifndef USING_CELL
    dequeue(2);
    #endif

    calc_distance_core(distance_block, Hdccf, psr_block, image);

    num_Distances = 1;
    enqueue_output(0, Distances_data_);

    return_buffer(Img_data_);
    return_buffer(PSRs_data_);
    return_buffer(HDCCF_Filters_data_);
#ifdef USING_CELL
  }
#endif
} /* Calculate_Distance(void *data, int mode) */
```

APPENDIX F

GENERATED DATA TYPE HEADER FILE

```
#ifndef DATATYPESYSTEM_H
#define DATATYPESYSTEM_H

/*DatatypeSystem.h Generated on Wed Feb 18 11:33:49 2009
*/

typedef float Float;
typedef double Double;
typedef float CplxFloatImgArray[128*128*2];
typedef float HMACH[128*128*2];
typedef float HDCCF[(32*32*2+1)*4];
typedef short Short;
typedef int Int;
typedef long long Long;
typedef  unsigned char Uchar;
typedef  unsigned char CharArray[128*128];
typedef struct CIMAGE_struct {
  int packet_type;
  int rows;
  int cols;
  int block_number;
  int testing_class;
  float mean;
  float std;
  unsigned char pad[4];
  CplxFloatImgArray data;
} CIMAGE;

typedef struct RAW_IMAGE_struct {
  int packet_type;
  int rows;
  int cols;
  int block_number;
  int testing_class;
  float mean;
  float std;
  unsigned char pad[4];
  CharArray data;
} RAW_IMAGE;

typedef struct PSR_struct {
  int x;
  int y;
  int refno;
  float value;
} PSR;
```

```
typedef struct PSR_BLOCK_struct {
  int block_number;
  unsigned char pad[12];
  PSR psrs[8];
} PSR_BLOCK;

typedef struct DISTANCE_struct {
  int testing_class;
  int x;
  int y;
  float psr_val;
  float value[3];
  unsigned char pad[4];
} DISTANCE;

typedef struct DISTANCE_BLOCK_struct {
  int block_number;
  int testing_class;
  int num_distances;
  unsigned char pad[4];
  DISTANCE distances[8];
} DISTANCE_BLOCK;

typedef struct DISTANCE_TABLE_struct {
  int block_number;
  unsigned char pad[12];
  DISTANCE distances[3*8];
} DISTANCE_TABLE;

typedef struct TARGET_struct {
  int x;
  int y;
  unsigned char pad[8];
} TARGET;

typedef struct TARGET_BLOCK_struct {
  int block_number;
  unsigned char pad[12];
  TARGET targets[3];
} TARGET_BLOCK;

#endif
```

APPENDIX G

GENERATED IX86 MAIN.C SYSTEM SOURCE FILE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <active.h>


#define PATH_FILE "matlab_path.txt"

int _node_number = 0;
int SAMPLE_RATE = 0;

/*tbd bke hack*/
char path_string[256];
char tmpstr[250];

int main(int argc, char *argv[])
{
  int loop_count = 0;
  int count=0;
  int len;
  FILE *path_file;
  printf("Initting... \n");

  if((path_file = fopen(PATH_FILE , "rt")) == NULL)
  {
        printf("ERROR!! Couldn't open matlab path file <<%s>>!!",
PATH_FILE);
        exit(-1);
  }

  fgets(tmpstr, 245, path_file);
  tmpstr[245] = '\0';
  len = strlen(tmpstr);
  if(tmpstr[len -1] == '\n')
        tmpstr[len-1] = '\0';
  fclose(path_file);
  sprintf(path_string, "%s", tmpstr);

  printf("The path is %s\n", path_string);

  active_init();
  process_funcp_table_init();
  printf("  Done Initting \n");
  active_build_network("build.cfg");

  for(;;)
  {
```

```
    if((loop_count++ %10000) == 0)
      {
            printf("loop %d.\n", loop_count);
      }

    active_tick();
      if (kbhit())
    {
      break;
    }
  }
  return 0;
}
```

APPENDIX H

GENERATED IX86 SYSTEM MAKEFILE

```
ifndef ACS_ENV
acs-env-not-defined:
      @echo Set the environment variable by sourcing setup.$$HOSTNAME
endif


#
# Makefile for ix86 specific code
#

TOPDIR            =../../../../
PRJDIR            =../../
ARCH=ix86

all: Debug

include $(TOPDIR)/etc/Rules.gnu

Debug Release: dummy
      @MAKEFLAGS=; nmake /nologo /f active.mak "CFG=active - Win32 $@"

clean: dummy
      $(RM) *~ *.log TAGS

wipe distclean: dummy
      $(RM) *~ *.log TAGS
      $(RM) -r Debug Release
      $(RM) *.clw *.aps *.ncb *.opt *.plg
      $(RM) .depend.*
```

APPENDIX I

GENERATED PROCESS TABLE

```
#include "active.h"

extern void ImgSplitter(void *, int);
extern void Image_Convert(void *, int);
extern void TwoDFFT(void *, int);
extern void splitter(void *, int);
extern void Multiply(void *, int);
extern void TwoDIFFT(void *, int);
extern void Calc_Mean_Std(void *, int);
extern void Calc_PSR(void *, int);
extern void Calculate_Distance(void *, int);
extern void image_src(void *, int);
extern void hmach_filter_src(void *, int);
extern void hdccf_filter_src(void *, int);
extern void Merge_Compare_Distances(void *, int);
extern void Post_Processing(void *, int);
extern void Target_Overlay(void *, int);
extern void ui(void *, int);

void process_funcp_table_init() {
  switch (_node_number) {
    case 0: {
      process_funcp_table[  0] = ImgSplitter;
      process_funcp_table[  1] = Image_Convert;
      process_funcp_table[  2] = TwoDFFT;
      process_funcp_table[  3] = splitter;
      process_funcp_table[  4] = Multiply;
      process_funcp_table[  5] = TwoDIFFT;
      process_funcp_table[  6] = Calc_Mean_Std;
      process_funcp_table[  7] = Calc_PSR;
      process_funcp_table[  8] = Calculate_Distance;
      process_funcp_table[  9] = image_src;
      process_funcp_table[ 10] = hmach_filter_src;
      process_funcp_table[ 11] = hdccf_filter_src;
      process_funcp_table[ 12] = Merge_Compare_Distances;
      process_funcp_table[ 13] = Post_Processing;
      process_funcp_table[ 14] = Target_Overlay;
      process_funcp_table[ 15] = ui;
      break;
    }
  }
}

void bct_init() {
  switch (_node_number) {
    case 0: {
      manager_root_stream = stream_create(0, 3, -1, -1, -1);
```

```
      manager_children = 0;
      break;
    }
  }
}
```

APPENDIX J

GENERATED CONFIGURATION FILE

```
; install local stream[1] on hw[0]
17 0 1 3 -1 -1 -1
; install local stream[2] on hw[0]
17 0 2 3 -1 -1 -1
; install local stream[3] on hw[0]
17 0 3 3 -1 -1 -1
; install local stream[4] on hw[0]
17 0 4 3 -1 -1 -1
; install local stream[5] on hw[0]
17 0 5 3 -1 -1 -1
; install local stream[6] on hw[0]
17 0 6 3 -1 -1 -1
; install local stream[7] on hw[0]
17 0 7 3 -1 -1 -1
; install local stream[8] on hw[0]
17 0 8 3 -1 -1 -1
; install local stream[9] on hw[0]
17 0 9 3 -1 -1 -1
; install local stream[10] on hw[0]
17 0 10 3 -1 -1 -1
; install local stream[11] on hw[0]
17 0 11 3 -1 -1 -1
; install local stream[12] on hw[0]
17 0 12 3 -1 -1 -1
; install local stream[13] on hw[0]
17 0 13 3 -1 -1 -1
; install local stream[14] on hw[0]
17 0 14 3 -1 -1 -1
; install local stream[15] on hw[0]
17 0 15 3 -1 -1 -1
; install local stream[16] on hw[0]
17 0 16 3 -1 -1 -1
; install local stream[17] on hw[0]
17 0 17 3 -1 -1 -1
; install local stream[18] on hw[0]
17 0 18 3 -1 -1 -1
; install local stream[19] on hw[0]
17 0 19 3 -1 -1 -1
; install local stream[20] on hw[0]
17 0 20 3 -1 -1 -1
; install local stream[21] on hw[0]
17 0 21 3 -1 -1 -1
; install local stream[22] on hw[0]
17 0 22 3 -1 -1 -1
; install local stream[23] on hw[0]
17 0 23 3 -1 -1 -1
; install local stream[24] on hw[0]
```

```
17 0 24 3 -1 -1 -1
; install local stream[25] on hw[0]
17 0 25 3 -1 -1 -1
; install local stream[26] on hw[0]
17 0 26 3 -1 -1 -1
; install local stream[27] on hw[0]
17 0 27 3 -1 -1 -1
; install local stream[28] on hw[0]
17 0 28 3 -1 -1 -1
; install local stream[29] on hw[0]
17 0 29 3 -1 -1 -1
; install local stream[30] on hw[0]
17 0 30 3 -1 -1 -1
; install local stream[31] on hw[0]
17 0 31 3 -1 -1 -1
; install local stream[32] on hw[0]
17 0 32 3 -1 -1 -1
; install local stream[33] on hw[0]
17 0 33 3 -1 -1 -1
; install local stream[34] on hw[0]
17 0 34 3 -1 -1 -1
; install local stream[35] on hw[0]
17 0 35 3 -1 -1 -1
; install local stream[36] on hw[0]
17 0 36 3 -1 -1 -1
; install local stream[37] on hw[0]
17 0 37 3 -1 -1 -1
; install local stream[38] on hw[0]
17 0 38 3 -1 -1 -1
; install local stream[39] on hw[0]
17 0 39 3 -1 -1 -1
; install local stream[40] on hw[0]
17 0 40 3 -1 -1 -1
; install sw[0] on hw[0]: script[0] (ImgSplitter), inputs[1],
outputs[3] param_words[0]
4 0 0 0 1 3 0
; install sw[1] on hw[0]: script[1] (Image_Convert), inputs[1],
outputs[1] param_words[0]
4 0 1 1 1 1 0
; install sw[2] on hw[0]: script[0] (ImgSplitter), inputs[1],
outputs[3] param_words[0]
4 0 2 0 1 3 0
; install sw[3] on hw[0]: script[2] (TwoDFFT), inputs[1], outputs[1]
param_words[0]
4 0 3 2 1 1 0
; install sw[4] on hw[0]: script[3] (splitter), inputs[1], outputs[2]
param_words[0]
4 0 4 3 1 2 0
; install sw[5] on hw[0]: script[4] (Multiply), inputs[2], outputs[1]
param_words[0]
4 0 5 4 2 1 0
; install sw[6] on hw[0]: script[5] (TwoDIFFT), inputs[1], outputs[1]
param_words[0]
4 0 6 5 1 1 0
; install sw[7] on hw[0]: script[6] (Calc_Mean_Std), inputs[1],
outputs[1] param_words[0]
4 0 7 6 1 1 0
```

```
; install sw[8] on hw[0]: script[7] (Calc_PSR), inputs[1], outputs[1]
param_words[0]
4 0 8 7 1 1 0
; install sw[9] on hw[0]: script[8] (Calculate_Distance), inputs[3],
outputs[1] param_words[0]
4 0 9 8 3 1 0
; install sw[10] on hw[0]: script[4] (Multiply), inputs[2], outputs[1]
param_words[0]
4 0 10 4 2 1 0
; install sw[11] on hw[0]: script[5] (TwoDIFFT), inputs[1], outputs[1]
param_words[0]
4 0 11 5 1 1 0
; install sw[12] on hw[0]: script[6] (Calc_Mean_Std), inputs[1],
outputs[1] param_words[0]
4 0 12 6 1 1 0
; install sw[13] on hw[0]: script[7] (Calc_PSR), inputs[1], outputs[1]
param_words[0]
4 0 13 7 1 1 0
; install sw[14] on hw[0]: script[8] (Calculate_Distance), inputs[3],
outputs[1] param_words[0]
4 0 14 8 3 1 0
; install sw[15] on hw[0]: script[4] (Multiply), inputs[2], outputs[1]
param_words[0]
4 0 15 4 2 1 0
; install sw[16] on hw[0]: script[5] (TwoDIFFT), inputs[1], outputs[1]
param_words[0]
4 0 16 5 1 1 0
; install sw[17] on hw[0]: script[6] (Calc_Mean_Std), inputs[1],
outputs[1] param_words[0]
4 0 17 6 1 1 0
; install sw[18] on hw[0]: script[7] (Calc_PSR), inputs[1], outputs[1]
param_words[0]
4 0 18 7 1 1 0
; install sw[19] on hw[0]: script[8] (Calculate_Distance), inputs[3],
outputs[1] param_words[0]
4 0 19 8 3 1 0
; install sw[20] on hw[0]: script[9] (image_src), inputs[0], outputs[3]
param_words[0]
4 0 20 9 0 3 0
; install sw[21] on hw[0]: script[10] (hmach_filter_src), inputs[1],
outputs[3] param_words[0]
4 0 21 10 1 3 0
; install sw[22] on hw[0]: script[11] (hdccf_filter_src), inputs[1],
outputs[1] param_words[0]
4 0 22 11 1 1 0
; install sw[23] on hw[0]: script[12] (Merge_Compare_Distances),
inputs[3], outputs[1] param_words[0]
4 0 23 12 3 1 0
; install sw[24] on hw[0]: script[13] (Post_Processing), inputs[1],
outputs[1] param_words[0]
4 0 24 13 1 1 0
; install sw[25] on hw[0]: script[14] (Target_Overlay), inputs[2],
outputs[1] param_words[0]
4 0 25 14 2 1 0
; install sw[26] on hw[0]: script[15] (ui), inputs[1], outputs[0]
param_words[0]
4 0 26 15 1 0 0
```

```
; install sw[27] on hw[0]: script[3] (splitter), inputs[1], outputs[2]
param_words[0]
4 0 27 3 1 2 0
; install sw[28] on hw[0]: script[3] (splitter), inputs[1], outputs[3]
param_words[0]
4 0 28 3 1 3 0
; connect input port[0] of sw[0] to stream[18] on hw[0]
11 0 0 0 18
; connect output port[0] of sw[0] to stream[1] on hw[0]
10 0 0 0 1
; connect output port[1] of sw[0] to stream[2] on hw[0]
10 0 0 1 2
; connect output port[2] of sw[0] to stream[3] on hw[0]
10 0 0 2 3
; connect input port[0] of sw[1] to stream[13] on hw[0]
11 0 1 0 13
; connect output port[0] of sw[1] to stream[17] on hw[0]
10 0 1 0 17
; connect input port[0] of sw[2] to stream[19] on hw[0]
11 0 2 0 19
; connect output port[0] of sw[2] to stream[4] on hw[0]
10 0 2 0 4
; connect output port[1] of sw[2] to stream[5] on hw[0]
10 0 2 1 5
; connect output port[2] of sw[2] to stream[6] on hw[0]
10 0 2 2 6
; connect input port[0] of sw[3] to stream[20] on hw[0]
11 0 3 0 20
; connect output port[0] of sw[3] to stream[18] on hw[0]
10 0 3 0 18
; connect input port[0] of sw[4] to stream[17] on hw[0]
11 0 4 0 17
; connect output port[0] of sw[4] to stream[19] on hw[0]
10 0 4 0 19
; connect output port[1] of sw[4] to stream[20] on hw[0]
10 0 4 1 20
; connect input port[0] of sw[5] to stream[1] on hw[0]
11 0 5 0 1
; connect input port[1] of sw[5] to stream[11] on hw[0]
11 0 5 1 11
; connect output port[0] of sw[5] to stream[21] on hw[0]
10 0 5 0 21
; connect input port[0] of sw[6] to stream[21] on hw[0]
11 0 6 0 21
; connect output port[0] of sw[6] to stream[22] on hw[0]
10 0 6 0 22
; connect input port[0] of sw[7] to stream[22] on hw[0]
11 0 7 0 22
; connect output port[0] of sw[7] to stream[23] on hw[0]
10 0 7 0 23
; connect input port[0] of sw[8] to stream[23] on hw[0]
11 0 8 0 23
; connect output port[0] of sw[8] to stream[24] on hw[0]
10 0 8 0 24
; connect input port[0] of sw[9] to stream[4] on hw[0]
11 0 9 0 4
; connect input port[1] of sw[9] to stream[24] on hw[0]
```

```
11 0 9 1 24
; connect input port[2] of sw[9] to stream[15] on hw[0]
11 0 9 2 15
; connect output port[0] of sw[9] to stream[7] on hw[0]
10 0 9 0 7
; connect input port[0] of sw[10] to stream[2] on hw[0]
11 0 10 0 2
; connect input port[1] of sw[10] to stream[12] on hw[0]
11 0 10 1 12
; connect output port[0] of sw[10] to stream[25] on hw[0]
10 0 10 0 25
; connect input port[0] of sw[11] to stream[25] on hw[0]
11 0 11 0 25
; connect output port[0] of sw[11] to stream[26] on hw[0]
10 0 11 0 26
; connect input port[0] of sw[12] to stream[26] on hw[0]
11 0 12 0 26
; connect output port[0] of sw[12] to stream[27] on hw[0]
10 0 12 0 27
; connect input port[0] of sw[13] to stream[27] on hw[0]
11 0 13 0 27
; connect output port[0] of sw[13] to stream[28] on hw[0]
10 0 13 0 28
; connect input port[0] of sw[14] to stream[5] on hw[0]
11 0 14 0 5
; connect input port[1] of sw[14] to stream[28] on hw[0]
11 0 14 1 28
; connect input port[2] of sw[14] to stream[14] on hw[0]
11 0 14 2 14
; connect output port[0] of sw[14] to stream[8] on hw[0]
10 0 14 0 8
; connect input port[0] of sw[15] to stream[3] on hw[0]
11 0 15 0 3
; connect input port[1] of sw[15] to stream[10] on hw[0]
11 0 15 1 10
; connect output port[0] of sw[15] to stream[29] on hw[0]
10 0 15 0 29
; connect input port[0] of sw[16] to stream[29] on hw[0]
11 0 16 0 29
; connect output port[0] of sw[16] to stream[30] on hw[0]
10 0 16 0 30
; connect input port[0] of sw[17] to stream[30] on hw[0]
11 0 17 0 30
; connect output port[0] of sw[17] to stream[31] on hw[0]
10 0 17 0 31
; connect input port[0] of sw[18] to stream[31] on hw[0]
11 0 18 0 31
; connect output port[0] of sw[18] to stream[32] on hw[0]
10 0 18 0 32
; connect input port[0] of sw[19] to stream[6] on hw[0]
11 0 19 0 6
; connect input port[1] of sw[19] to stream[32] on hw[0]
11 0 19 1 32
; connect input port[2] of sw[19] to stream[16] on hw[0]
11 0 19 2 16
; connect output port[0] of sw[19] to stream[9] on hw[0]
10 0 19 0 9
```

```
; connect output port[2] of sw[20] to stream[33] on hw[0]
10 0 20 2 33
; connect output port[0] of sw[20] to stream[34] on hw[0]
10 0 20 0 34
; connect output port[1] of sw[20] to stream[35] on hw[0]
10 0 20 1 35
; connect input port[0] of sw[21] to stream[35] on hw[0]
11 0 21 0 35
; connect output port[2] of sw[21] to stream[10] on hw[0]
10 0 21 2 10
; connect output port[0] of sw[21] to stream[11] on hw[0]
10 0 21 0 11
; connect output port[1] of sw[21] to stream[12] on hw[0]
10 0 21 1 12
; connect input port[0] of sw[22] to stream[33] on hw[0]
11 0 22 0 33
; connect output port[0] of sw[22] to stream[36] on hw[0]
10 0 22 0 36
; connect input port[0] of sw[23] to stream[7] on hw[0]
11 0 23 0 7
; connect input port[1] of sw[23] to stream[8] on hw[0]
11 0 23 1 8
; connect input port[2] of sw[23] to stream[9] on hw[0]
11 0 23 2 9
; connect output port[0] of sw[23] to stream[37] on hw[0]
10 0 23 0 37
; connect input port[0] of sw[24] to stream[37] on hw[0]
11 0 24 0 37
; connect output port[0] of sw[24] to stream[38] on hw[0]
10 0 24 0 38
; connect input port[0] of sw[25] to stream[40] on hw[0]
11 0 25 0 40
; connect input port[1] of sw[25] to stream[38] on hw[0]
11 0 25 1 38
; connect output port[0] of sw[25] to stream[39] on hw[0]
10 0 25 0 39
; connect input port[0] of sw[26] to stream[39] on hw[0]
11 0 26 0 39
; connect input port[0] of sw[27] to stream[34] on hw[0]
11 0 27 0 34
; connect output port[0] of sw[27] to stream[40] on hw[0]
10 0 27 0 40
; connect output port[1] of sw[27] to stream[13] on hw[0]
10 0 27 1 13
; connect input port[0] of sw[28] to stream[36] on hw[0]
11 0 28 0 36
; connect output port[1] of sw[28] to stream[14] on hw[0]
10 0 28 1 14
; connect output port[0] of sw[28] to stream[15] on hw[0]
10 0 28 0 15
; connect output port[2] of sw[28] to stream[16] on hw[0]
10 0 28 2 16
; activate sw[0] on hw[0]
8 0 0
; activate sw[1] on hw[0]
8 0 1
; activate sw[2] on hw[0]
```

```
8 0 2
; activate sw[3] on hw[0]
8 0 3
; activate sw[4] on hw[0]
8 0 4
; activate sw[5] on hw[0]
8 0 5
; activate sw[6] on hw[0]
8 0 6
; activate sw[7] on hw[0]
8 0 7
; activate sw[8] on hw[0]
8 0 8
; activate sw[9] on hw[0]
8 0 9
; activate sw[10] on hw[0]
8 0 10
; activate sw[11] on hw[0]
8 0 11
; activate sw[12] on hw[0]
8 0 12
; activate sw[13] on hw[0]
8 0 13
; activate sw[14] on hw[0]
8 0 14
; activate sw[15] on hw[0]
8 0 15
; activate sw[16] on hw[0]
8 0 16
; activate sw[17] on hw[0]
8 0 17
; activate sw[18] on hw[0]
8 0 18
; activate sw[19] on hw[0]
8 0 19
; activate sw[20] on hw[0]
8 0 20
; activate sw[21] on hw[0]
8 0 21
; activate sw[22] on hw[0]
8 0 22
; activate sw[23] on hw[0]
8 0 23
; activate sw[24] on hw[0]
8 0 24
; activate sw[25] on hw[0]
8 0 25
; activate sw[26] on hw[0]
8 0 26
; activate sw[27] on hw[0]
8 0 27
; activate sw[28] on hw[0]
8 0 28
; start hw[0]
1 0
```

APPENDIX K

GENERATED CELL MAIN.C SYSTEM SOURCE FILE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <active.h>

int _node_number = 0;
int SAMPLE_RATE = 0;

int main(int argc, char *argv[])
{
  int loop_count = 0;
  int count=0;
  int len;


  printf("\n\n\n  Starting APP \n");
  active_init();
  process_funcp_table_init();
  active_build_network("build.cfg");
  printf("  Done Initting \n");

  for(;;)
  {
  }
  return 0;
}
```

APPENDIX L

GENERATED CELL SYSTEM MAKEFILE

```
ifndef ACS_ENV
acs-env-not-defined:
      @echo Set the environment variable by sourcing setup.$$HOSTNAME
endif


#
# Makefile for fc6-cbe specific code
#

TOPDIR            = ../../../..
PRJDIR            = ../..

Debug Release: dummy
      [ -d $@ ] || mkdir $@
      $(MAKE) -f Makefile CFG=$@ ARCH=cbe All

_LIBS = $(shell ls $(PRJDIR)/sw_component/$(CFG))
LIBS  = $(addprefix $(PRJDIR)/sw_component/$(CFG)/, $(_LIBS))

ifdef CFG
INTDIR            = $(CFG)

VPATH       =
../:$(TOPDIR)/runtime/kernel/:$(TOPDIR)/runtime/arch/common/:$(TOPDIR)/
runtime/arch/fc6-
cbe/:$(PRJDIR)/sw_component/:$(PRJDIR)/sw_component/$(INTDIR)

CFLAGS            = -I $(TOPDIR)/include -I $(PRJDIR)/include -qstrict
-qfloat=rsqrt

IMPORTS           = -lspe2 -lpthread -lm -lfft
$(TOPDIR)/lib/libkernel.a $(TOPDIR)/lib/lowcbe.a
$(PRJDIR)/sw_component/$(CFG)/proclib.a $(filter %.a, $(LIBS))

CC_OPT_LEVEL      = -O5

OBJS        = init.o main.o

ifeq ($(CFG),Debug)
CFLAGS      += -pg -DDEBUG
OBJS += debug.o
endif

PROGRAM_ppu = active.exe

ifeq ($(CFG),Debug)
```

```
PROGRAM_ppu = active_d.exe
endif

TARGET_INSTALL_DIR = $(PRJDIR)/run
INSTALL_FILES     = $(OBJS)
INSTALL_DIR = $(INTDIR)

endif

include $(TOPDIR)/etc/Rules.gnu

All:
      $(MAKE) -f Makefile CFG=$(INTDIR) ARCH=cbe all
ifdef _USING_XLC
      $(CP) $(patsubst %.o, %.d, $(OBJS)) $(INTDIR)
endif
      $(RM) $(INSTALL_FILES) *.a *.exe

ifneq ($(ARCH),cbe)
clean:
      $(MAKE) -f Makefile CFG=$(INTDIR) ARCH=cbe clean
endif

wipe: dummy
      $(RM) -fr Debug Release
      $(MAKE) -f Makefile CFG=$(INTDIR) ARCH=cbe clean
```

APPENDIX M

GENERATED WRAPPER CODE FOR SPE-RUN CALCULATE_DISTANCE

```
/* AWCalculate_Distance.c generated on Tue Feb 24 10:54:13 2009
 */

#include <active.h>
#include "DatatypeSystem.h"

#include <libspe2.h>

extern spe_program_handle_t calc_distance_core_spu;

typedef struct _calc_distance_core_context_data_t
{
  CIMAGE *Img_ea;
  PSR_BLOCK *PSRs_ea;
  HDCCF *HDCCF_Filters_ea;
  DISTANCE_BLOCK *Distances_ea;
  unsigned char pad[(16-(sizeof(CIMAGE*) + sizeof(PSR_BLOCK*) +
sizeof(HDCCF*) + sizeof(DISTANCE_BLOCK*) + 0)%16)];
} calc_distance_core_context_data_t;


void Calculate_Distance(void *__data, int __mode)
{
  CIMAGE *Img_data_;
  PSR_BLOCK *PSRs_data_;
  HDCCF *HDCCF_Filters_data_;
  DISTANCE_BLOCK *Distances_data_;
  int num_Distances;

  spe_context_ptr_t ctx;
  unsigned int entry;
  spe_stop_info_t stop_info;
  calc_distance_core_context_data_t ctx_data __attribute__
((aligned(128)));

  ctx = spe_context_create(0, NULL);
  spe_program_load(ctx, &calc_distance_core_spu);

#ifdef USING_CELL
  for(;;)
  {
#endif
    #ifdef USING_CELL
    Img_data_ = (CIMAGE*)dequeue(0);
    #else
    Img_data_ = (CIMAGE*)get_input_buffer(0);
    #endif
```

```
    #ifdef USING_CELL
    PSRs_data_ = (PSR_BLOCK*)dequeue(1);
    #else
    PSRs_data_ = (PSR_BLOCK*)get_input_buffer(1);
    #endif
    #ifdef USING_CELL
    HDCCF_Filters_data_ = (HDCCF*)dequeue(2);
    #else
    HDCCF_Filters_data_ = (HDCCF*)get_input_buffer(2);
    #endif

    #ifdef USING_CELL
    if( !(Img_data_ && PSRs_data_ && HDCCF_Filters_data_) )
    {
      /* no data to execute or no room to put results */
      continue;
    }
    #else
    if( !(Img_data_ && PSRs_data_ && HDCCF_Filters_data_ &&
output_slot_available(0) ) )
    {
      /* no data to execute or no room to put results */
      return;
    }
    #endif

    Distances_data_ = (DISTANCE_BLOCK *)get_buffer(1 *
sizeof(DISTANCE_BLOCK)/sizeof(int));

    #ifdef USING_CELL
    if( !Distances_data_ )
    {
      /* bad output pointer */
      active_error(__LINE__, ACT_MEMORY_OUTOFMEMORY);
      continue;
    }
    #else
    if( !Distances_data_ )
    {
      /* bad output pointer */
      active_error(__LINE__, ACT_MEMORY_OUTOFMEMORY);
      return;
    }
    #endif

    #ifndef USING_CELL
    dequeue(0);
    #endif
    #ifndef USING_CELL
    dequeue(1);
    #endif
    #ifndef USING_CELL
    dequeue(2);
    #endif

    entry = SPE_DEFAULT_ENTRY;
    ctx_data.Img_ea = Img_data_;
```

```
    ctx_data.PSRs_ea = PSRs_data_;
    ctx_data.HDCCF_Filters_ea = HDCCF_Filters_data_;
    ctx_data.Distances_ea = Distances_data_;

    spe_context_run(ctx, &entry, 0, &ctx_data, NULL, &stop_info);

    enqueue_output(0, Distances_data_);

    return_buffer(Img_data_);
    return_buffer(PSRs_data_);
    return_buffer(HDCCF_Filters_data_);
#ifdef USING_CELL
  }
#endif
} /* Calculate_Distance(void *data, int mode) */
```

APPENDIX N

GENERATED SPE CODE FOR CALCULATE_DISTANCE

```
/* calc_distance_core_spu.c generated on Tue Feb 24 10:54:13 2009
 */

#include <active.h>
#include "DatatypeSystem.h"

#include <spu_mfcio.h>
#include <free_align.h>
#include <malloc_align.h>

extern void  calc_distance_core(DISTANCE_BLOCK *distance_block, float
*Hdccf, PSR_BLOCK *psr_block, CIMAGE *image);

typedef struct _calc_distance_core_context_data_t
{
  CIMAGE *Img_ea;
  PSR_BLOCK *PSRs_ea;
  HDCCF *HDCCF_Filters_ea;
  DISTANCE_BLOCK *Distances_ea;
  unsigned char pad[(16-(sizeof(CIMAGE*) + sizeof(PSR_BLOCK*) +
sizeof(HDCCF*) + sizeof(DISTANCE_BLOCK*) + 0)%16)];
} calc_distance_core_context_data_t;

void main(unsigned long long speid, addr64 argp, addr64 envp)
{
  int i, j, dma_size;

  CIMAGE *Img_data_;
  PSR_BLOCK *PSRs_data_;
  PSR_BLOCK _PSRs_data __attribute__ ((aligned(128)));
  HDCCF *HDCCF_Filters_data_;
  HDCCF _HDCCF_Filters_data __attribute__ ((aligned(128)));
  DISTANCE_BLOCK *Distances_data_;
  DISTANCE_BLOCK _Distances_data __attribute__ ((aligned(128)));
  int num_Distances;

  CIMAGE *image;
  PSR_BLOCK *psr_block;
  float *Hdccf;
  DISTANCE_BLOCK *distance_block;

  calc_distance_core_context_data_t ctx_data __attribute__
((aligned(128)));

  addr64 Img_addr;
  addr64 PSRs_addr;
  addr64 HDCCF_Filters_addr;
```

```
  addr64 Distances_addr;

  Img_data_ = (CIMAGE *)_malloc_align(sizeof(CIMAGE), 7);
  PSRs_data_ = &_PSRs_data;
  HDCCF_Filters_data_ = &_HDCCF_Filters_data;
  Distances_data_ = &_Distances_data;

  if( !Img_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  if( !PSRs_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  if( !HDCCF_Filters_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  if( !Distances_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  mfc_get(&ctx_data, argp.ull,
sizeof(calc_distance_core_context_data_t), 31, 0, 0);
  mfc_write_tag_mask(1<<31);
  mfc_read_tag_status_all();

  Img_addr.ull = (unsigned int)ctx_data.Img_ea;
  PSRs_addr.ull = (unsigned int)ctx_data.PSRs_ea;
  HDCCF_Filters_addr.ull = (unsigned int)ctx_data.HDCCF_Filters_ea;
  Distances_addr.ull = (unsigned int)ctx_data.Distances_ea;

  dma_size = sizeof(CIMAGE);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)Img_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Img_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_get((int*)Img_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Img_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();
```

```
  dma_size = sizeof(PSR_BLOCK);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)PSRs_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
PSRs_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_get((int*)PSRs_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
PSRs_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();
  dma_size = sizeof(HDCCF);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)HDCCF_Filters_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
HDCCF_Filters_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0,
0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_get((int*)HDCCF_Filters_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
HDCCF_Filters_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();


  image = Img_data_;
  psr_block = PSRs_data_;
  Hdccf = HDCCF_Filters_data_;

  distance_block = Distances_data_;

  calc_distance_core(distance_block, Hdccf, psr_block, image);


  num_Distances = 1;

  dma_size = sizeof(DISTANCE_BLOCK);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_put((int*)Distances_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Distances_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_put((int*)Distances_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Distances_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();

  _free_align(Img_data_);

} /* calc_distance_core SPE Core */
```

APPENDIX O

GENERATED SPE MAKEFILE FOR CALCULATE_DISTANCE

```
## calc_distance_core Makefile generated on Tue Feb 24 10:54:13 2009

## You need to add flags to IMPORTS and any other desired adds

TOPDIR            = ../../../..
PRJDIR            = ../..
ARCH              = cbe

PROGRAM_spu       = calc_distance_core_spu
LIBRARY_embed     = calc_distance_core_spu.a

VPATH             = ../
OBJS              = calc_distance_core_spu.o  calc_distance_core.o
fft_core.o
CC_OPT_LEVEL      = -O5
CFLAGS            = -I $(TOPDIR)/include -I $(PRJDIR)/include
IMPORTS           = -lm
INSTALL_FILES     = calc_distance_core_spu.a
INSTALL_DIR       = ..

include $(TOPDIR)/etc/Rules.gnu
```

APPENDIX P

MODIFIED SPE CODE FOR MULT_DIFFT_CALC_MEAN_PSR

```
/* mult_difft_calc_mean_psr_core_spu.c generated on Wed Feb 18 15:25:53
2009
 */

#include <active.h>
#include "DatatypeSystem.h"

#include <spu_mfcio.h>
#include <free_align.h>
#include <malloc_align.h>

extern void  mult_difft_calc_mean_psr_core(float *Hmach, CIMAGE *img,
PSR_BLOCK *psrs);

typedef struct _mult_difft_calc_mean_psr_core_context_data_t
{
  HMACH *Filter_ea;
  CIMAGE *Spectrum_ea;
  PSR_BLOCK *PSRs_ea;
  unsigned char pad[(16-(sizeof(HMACH*) + sizeof(CIMAGE*) +
sizeof(PSR_BLOCK*) + 0)%16)];
} mult_difft_calc_mean_psr_core_context_data_t;

void main(unsigned long long speid, addr64 argp, addr64 envp)
{
  int i, j, dma_size;

  HMACH *Filter_data_;
  CIMAGE *Spectrum_data_;
  CIMAGE _Spectrum_data __attribute__ ((aligned(128)));
  PSR_BLOCK *PSRs_data_;
  PSR_BLOCK _PSRs_data __attribute__ ((aligned(128)));
  int num_PSRs;

  CIMAGE *img;
  float *Hmach;
  PSR_BLOCK *psrs;

  mult_difft_calc_mean_psr_core_context_data_t ctx_data __attribute__
((aligned(128)));

  addr64 Filter_addr;
  addr64 Spectrum_addr;
  addr64 PSRs_addr;

  Filter_data_ = (HMACH *)_malloc_align(sizeof(HMACH)/2, 7);
  Spectrum_data_ = &_Spectrum_data;
```

```
  PSRs_data_ = &_PSRs_data;

  if( !Filter_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  if( !Spectrum_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  if( !PSRs_data_ )
  {
    /* bad output pointer */
    printf("Failed Malloc");
    return;
  }

  mfc_get(&ctx_data, argp.ull,
sizeof(mult_difft_calc_mean_psr_core_context_data_t), 31, 0, 0);
  mfc_write_tag_mask(1<<31);
  mfc_read_tag_status_all();

  Filter_addr.ull = (unsigned int)ctx_data.Filter_ea;
  Spectrum_addr.ull = (unsigned int)ctx_data.Spectrum_ea;
  PSRs_addr.ull = (unsigned int)ctx_data.PSRs_ea;

  dma_size = sizeof(HMACH)/2;
  i=0;
  j=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)Filter_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Filter_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    j++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_get((int*)Filter_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Filter_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();
  j++;
  dma_size = sizeof(CIMAGE);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)Spectrum_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Spectrum_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
```

```
  mfc_get((int*)Spectrum_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Spectrum_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();


  img = Spectrum_data_;
  Hmach = Filter_data_;

  psrs = PSRs_data_;


  multiply_filter_core(img, Hmach, 0);

  dma_size = sizeof(HMACH)/2;
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_get((int*)Filter_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Filter_addr.ull + j*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    j++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_get((int*)Filter_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
Filter_addr.ull + j*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();


  mult_difft_calc_mean_psr_core(Hmach, img, psrs);

  num_PSRs = 1;

  dma_size = sizeof(PSR_BLOCK);
  i=0;
  while(dma_size > MFC_MAX_DMA_SIZE)
  {
    mfc_put((int*)PSRs_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
PSRs_addr.ull + i*MFC_MAX_DMA_SIZE, MFC_MAX_DMA_SIZE, 31, 0, 0);
    i++;
    dma_size -= MFC_MAX_DMA_SIZE;
  }
  mfc_put((int*)PSRs_data_ + i*MFC_MAX_DMA_SIZE/sizeof(int),
PSRs_addr.ull + i*MFC_MAX_DMA_SIZE, dma_size, 31, 0, 0);
  mfc_read_tag_status_all();

  _free_align(Filter_data_);

} /* mult_difft_calc_mean_psr_core SPE Core */
```

REFERENCES

[1] Asaad, S. Bapty, T. Neema, S. “Performance Modeling for Adaptive Parallel Embedded Systems.” *Performance, Computing, and Communications Conference, 2002. 21st IEEE International.* pp. 57-64. 3-5 April 2002.

[2] Chen, Long; Hu, Ziang; Lin, Junmin; Gao, G.R., "Optimizing the Fast Fourier Transform on a Multi-core Architecture," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* , pp.1-8, 26-30 March 2007.

[3] *GCC, the GNU Compiler Collection*. Free Software Foundation, Inc., 2009. [Online]. Available: http://gcc.gnu.org/. [Accessed: Jan. 30, 2009].

[4] Goodman, R.; Black, S., "Design challenges for realization of the advantages of embedded multi-core processors," *AUTOTESTCON, 2008 IEEE* , pp.447-452, 8-11 Sept. 2008.

[5] IBM. *Cell Broadband Engine Programming Handbook*, Version 1.1, April 24, 2007.

[6] IBM. *Cell Broadband Engine Programming Tutorial*, Version 2.1, March 1, 2007.

[7] IBM. *Cell Broadband Engine SDK Libraries Overview and Usrs Guide, Version 2.1*. 26 Mar. 2007.

[8] IBM. *Software Development Kit 2.1 Installation Guide, Version 2.1*. Mar. 2007.

[9] IBM. *SPE Runtime Management Library Version 2.1*. Mar. 2007.

[10] IBM. “The Cell Chip,” *The Cell Project at IBM Research*, 2006. [Online]. Available: http://www.research.ibm.com/cell/cell_chip.html. [Accessed: Jan. 27, 2009].

[11] IBM. *XL C/C++ for Multicore Acceleration for Linux*. [Online]. Available: http://www-01.ibm.com/software/awdtools/xlcpp/multicore/. [Accessed: Jan. 30, 2009]

[12] Karsai G., Sztipanovits J., Ledeczi A., Bapty T.: “Model-Integrated Development of Embedded Software,” *Proceedings of the IEEE*, Vol. 91, Number 1, pp. 145-164, January, 2003.

[13] Kayi, A.; Yao, Y.; El-Ghazawi, T.; Newby, G., "Experimental Evaluation of Emerging Multi-core Architectures," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* , pp.1-6, 26-30 March 2007.

[14] Kurzak, J., Buttari, A., Luszczek, P., Dongarra, J., “The PlayStation 3 for High-Performance Scientific Computing,” *Computing in Science & Engineering*, Vol. 10, Issue 3, pp. 84-87, May-June 2008.

[15] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P., “The Generic Modeling Environment,” *WISP’2001 Proceedings, IEEE*. 24-25 May 2001.

[16] Narayanaswamy, G.; Balaji, P.; Feng, W., "Impact of Network Sharing in Multi-Core Architectures," *Computer Communications and Networks, 2008. ICCCN '08. Proceedings of 17th International Conference on* , pp.1-6, 3-7 Aug. 2008.

[17] Neema, S., Bapty, T., Scott, J., Eames, B., “Signal processing platform: a tool chain for designing high performance signal processing applications,” *SoutheastCon, 2005. Proceedings. IEEE.* pp. 302-307, 8-10 April 2005

[18] Neema S., Sztipanovits J., Karsai G., and Butts K., “Constraint-based Design-Space Exploration and Model Synthesis,” *Proceedings of Embedded Software Conference (EMSOFT), 2003, in Lecture Notes in Computer Science (LNCS),* Springer-Verlag 2003.

[19] Nemati, F.; Kraft, J.; Nolte, T., "Towards migrating legacy real-time systems to multi-core platforms," *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on* , pp.717-720, 15-18 Sept. 2008

[20] Szydlowski, C. “Multithreaded Technology & Multicore Processors.” *Dr. Dobb’s Journal*, May 2005.

[21] The Fedora Project. See http://fedoraproject.org/ for more information.

[22] Xiang, Yang; Zhou, Wanlei, "Using Multi-Core Processors to Support Network Security Applications," *Future Trends of Distributed Computing Systems, 2008*. *FTDCS '08. 12th IEEE International Workshop on* , pp.213-218, 21-23 Oct. 2008.