Runtime Parallelization switching for MPEG4 encoder on MPSOC

NAEEM ABBAS

ROYAL INSTITUTE
OF TECHNOLOGY

Degree project in

Second cycle
Stockholm, Sweden 2012

TRITA-ICT-EX-2010:5

www.kth.se

ACKNOWLEDGEMENTS

First, I would like to Thank All mighty ALLAH for each and everything. I would like to thank Dr. Zhe Ma and Professor Ahmed Hemani for giving me the opportunity to work on a M.Sc. Thesis at IMEC and for their patience with us.

I would also like to thank Prabhat Avasare, Rogier Baert, Chantal Couvreur and all run-time team staff for their help at different phases throughout the Thesis work. I am thankful to IMEC, for providing excellent working and learning environment and offering opportunity to work on professional and close to industry problem.

Lastly, I would like to thank Government of Pakistan for giving me scholarship through NUST for pursuing master studies at KTH.

ABSTRACT

The recent development for multimedia applications on mobile terminals raised the need for flexible and scalable computing platforms that are capable of providing considerable (application specific) computational performance within a low cost and a low energy budget.

The MPSoC with multi-disciplinary approach, resolving application mapping, platform architecture and runtime management issues, provides such multiple heterogeneous, flexible processing elements. In MPSoC, the run-time manager takes the design time exploration information as an input and selects an active Pareto point based on quality requirement and available platform resources, where a Pareto point corresponds to a particular parallelization possibility of target application.

To use system's scalability at best and enhance application's flexibility a step further, the resource management and Pareto point selection decisions need to be adjustable at run-time. This thesis work experiments run-time Pareto point switching for MPEG-4 encoder. The work involves design time exploration and then embedding of two parallelization possibilities of MPEG-4 encoder into one single component and enabling run-time switching between parallelizations, to give run-time control over adjusting performance-cost criteria and allocation de-allocation of hardware resources at run-time. The newer system has the capability to encode each video frame with different parallelization. The obtained results offer a number of operating points on Pareto curve in between the previous ones at sequence encoding level. The run-time manager can improve application performance up to 50% or can save memory bandwidth up to 15%, according to quality request.

TABLE OF CONTENTS

ABSTRACT i
TABLE OF CONTENTS i
LIST OF FIGURES iii
ACKNOWLEDGEMENTS ..... **Error! Bookmark not defined.**
Chapter 1 Introduction 1
1.1 Background 1
1.1.1 Design Time Application Mapping 1
1.1.2 Platform Architecture Exploration 2
1.1.3 Run-time Platform Management 2
Chapter 2 MPSoC Programming Model 4
2.1 ADRES 5
2.1.1 ADRES Architecture 5
2.2 Memory Hierarchy 7
2.2.1 Scratch Pad Memories 7
2.2.2 THE MH TOOL 7
2.2.2.1 Data-reuse analysis 8
2.2.2.2 Optimization 8
2.3 The Run-time Manager 10
2.3.1 Quality Manager 10
2.3.2 Resource Manager 11
2.3.3 Run-Time Library 11
2.4 Clean C 12
2.5 ATOMIUM / ANALYSIS (ANL) 13
2.5.1 Atomium/Analysis Process Flow 14
2.6 MPA TOOL 15
2.7 High Level Simulator (HLSIM) 17
Chapter 3 Previous Work 19
3.1 Experimental workflow and code preparation 20
3.2 MPEG-4 Encoder Experiment 21
3.3 Platform Template 23
Chapter 4 Methodology & Implementation 25
4.1 ANL Profiling for Sequential MPEG-4 Encoder 26

4.1.1 Preparing the program for ATOMIUM 26
4.1.2 Instrumenting the program using ATOMIUM 27
4.1.3 Compiling and Linking the Instrumented program 27
4.1.4 Generating an Access count report 27
4.1.5 Interpreting results 27
4.2 Discussion on Previous Parallelizations 29
4.3 Choosing optimal parallelizations for run-time switching 31
4.4 Run-time parallelization switcher 32
4.4.1 Configuring new component 32
4.4.2 Switching decision spot 34
4.4.3 Manual adjustment to code and tools 36
4.4.3.1 Adding new threads and naming convention 36
4.4.3.2 Parsection switching & parallelization spawning 36
4.4.3.3 Replicating and Adjusting functionalities called by threads 37
4.4.3.4 Adjusting ARP instrumentation 38
4.4.3.5 Updating platform information 38
4.4.3.6 Block Transfers and FIFO communication 38
4.4.3.7 Execution and Profiling 38
Chapter 5 Results 40
5.1 Interpreting ANL profiling results 41
5.2 Profiling Parallelization switcher component 42
5.3 The Final Pareto Curve 45
Chapter 6 Conclusions and Future Work 47
6.1 Conclusions 47
6.2 Future Work 48
REFERENCES 49

LIST OF FIGURES

Figure 1 MPSoC – Combination of Disciplines[2] 1
Figure 2 - Run-time Management[4] 3
Figure 3 MPSoC Programming Model[5]. 4
Figure 4 Architecture of the ADRES Coarse Grain Reconfigurable Array[3] 5
Figure 5 - ADRES system (left) Reconfigurable Cell (right)[8] 6
Figure 6 MH overview[9]. 8
Figure 7 Scratch pad based platform architecture 9
Figure 8 the run-time manager[10]. 10
Figure 9 Atomium/Analysis Instrumentation Process[18] 13
Figure 10 Atomium/Analysis Generating Results[18] 14
Figure 11 MP-MH Tool Flow[19]. 15
Figure 12 Illustration of FIFO delay[19]. 17
Figure 13 Experimental tool flow[22]. 20
Figure 14 Top-level cycle distribution for 10 frame of crew 4CIF sequence[22]. 21
Figure 15 MPEG-4 Encoder parallelization options for various processor count[22] 22
Figure 16 MPEG-4 Encoder parallelization options for various processor count[22] 22
Figure 17 Platform Instantiation[22]. 23
Figure 18 Profiling Steps for ANL. 26
Figure 19 Execution Time Distribution for MPEG-4 Encoder (in microseconds) 28
Figure 20 Profiling results for proposed Parallelizations (crew 4CIF 10 frames sequence) 30
Figure 21 Optimal Parallelization Selection. 31
Figure 22 Parallelizations integration overview. 33
Figure 23 Component with both parallelizations. 33
Figure 24 MPEG-4 Encoder application control flow. 35
Figure 25 Parallelization spawning model 35
Figure 26 Total array accesses for selected functionalities (read + write). 41
Figure 27 Profiling Results for Parallelization Switching Component (Crew 4CIF sequence) 42
Figure 28 Switching parallelizations cost negligible 43
Figure 29 Profiling Results for Parallelization Switching Component (Harbor 4CIF sequence) 44
Figure 30 Performance-Cost Pareto Curve (Crew 4CIF 10-frame Sequence) 45
Figure 31 Performance-Cost Pareto Curve (Harbor 4CIF 10-frame Sequence) 46
Figure 32 The run-time manager[10] 47

Chapter 1 Introduction

1.1 Background

The multiprocessor system-on-chip introduced in market to cope with emerging market challenges regarding performance and power for mobile terminals. MPSoC is able to deliver aggressive performance at low power costs, but there are still some issues to be addressed before successful and efficient implementation of multimedia applications.

Firstly, currently it is the application designer's responsibility to map the application to the resources and services provided by platform. As the number of possibilities is huge, so best possible mapping will not be possible without a tool help.

Secondly, the platform architecture should meet application requirement as close as possible by providing right amount of service at right time.

Thirdly, in presence of multiple applications on platform, an arbiter is required to distribute and allocate resources between applications and reduce the interference. The arbiter will be working as run-time manager. The MPSoC activity is to combine these disciplines and produce effective solutions[1].

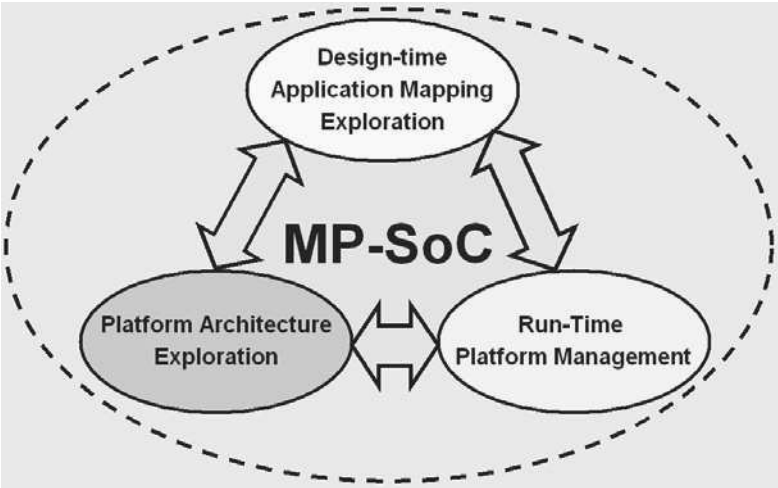

Figure 1 MPSoC – Combination of Disciplines[2]

1.1.1 Design Time Application Mapping

For design time application mapping, manual work may lead to sub-optimal solution. Moreover, with modification in platform or run-time circumstances, mapping needs to be done from scratch again. The design time exploration helps application designer, in form of methodologies and tools, to achieve optimal mapping solutions.

Design time exploration may cover many important issues about the system[1]:

- Estimation and Reservation of memory bandwidth
- Functional and data-level split exploration for parallelization opportunities.
- Exploration for shared data copy candidates finding data reuse and pre fetch.
- Exploration for instruction memory reuse possibilities

1.1.2 Platform Architecture Exploration

Platform architecture exploration takes task-oriented communication at application level and tries to find out solution on multiprocessor platform with memory hierarchy and interconnect network. The aim is to [3]:

- Identify required communication services by application and provide support to them.
- Optimize the communication mapping according to platform resources (e.g. NOC).
- Ensure run-time management for the communication services.

The run-time management is quite necessary as bandwidth allocation decision need to be done at run-time when new applications start. Fast heuristics are required to find possible route for data communication through NOC aiming low cost as well as guaranteed throughput.

1.1.3 Run-time Platform Management

The run-time manager works as a resource arbiter in presence of multiple applications on platform. During Design time exploration and profiling, several working Pareto points are defined for each application. While the run-time manager selects an active Pareto point for application based on required application quality. The exploration information at design time helps run-time manager, effectively adjust application's quality according to user requirement.

Currently, run-time manager is able to explore design time information regarding application at the start of run-time. When decision is taken and a Pareto point is selected for an application. The decision will be irreversible until the end of this execution. The resources once allocated to an application instance, it will be inaccessible until that instance release resources after execution.

The objective of this thesis work is to enable run-time switching between different Pareto points for an application according to change in user requirements at run-time. Ultimately, run-time manager will provide control over resource management at run-time and user will be able to tune the application quality desired from each application in presence of multiple running applications.

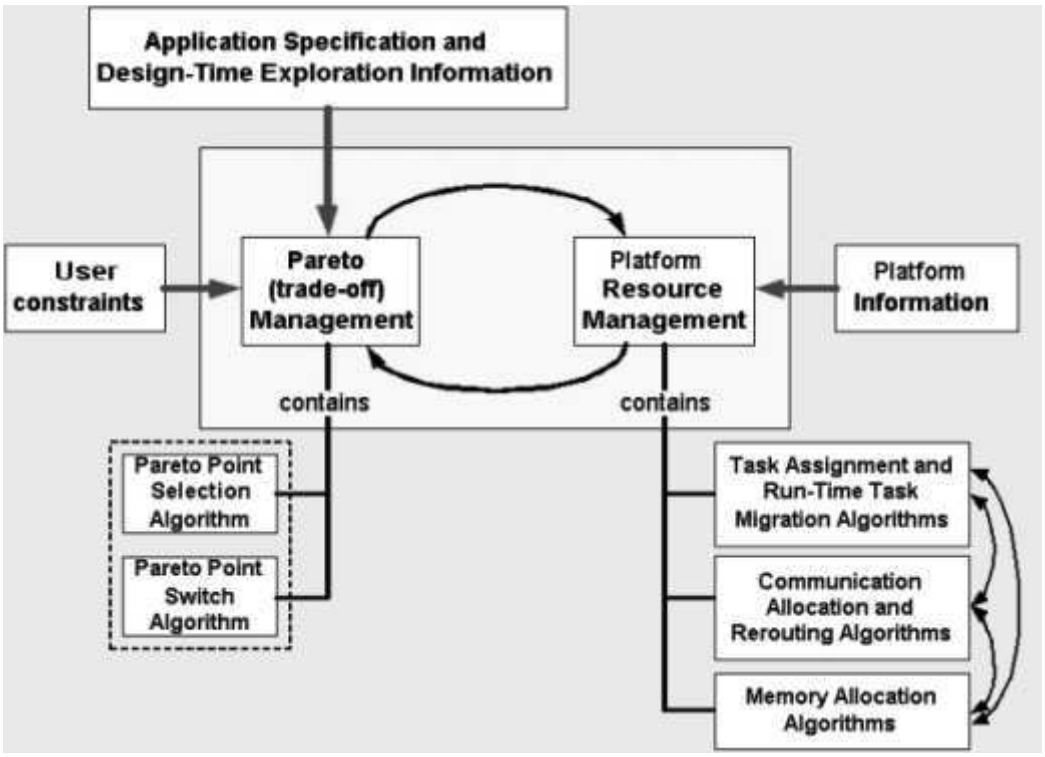

Figure 2 - Run-time Management[4]

The thesis work also includes the estimation of extra workload to tolerate for Pareto point switching at run-time. One also need to evaluate the defined Pareto point at design time and profile the efficiency of each Pareto points in terms of performance and bandwidth usage.

Chapter 2 MPSoC Programming Model

Using ordinary sequential code of any application and facilitating to run on multi-processor parallel execution environment needs many modification steps to follow. The ordinary sequential C code will be using full breadth of C environment, but some of them are not suitable for multiprocessor platform. First job is to detect, correct and follow such guidelines to make sequential code more compliant with hardware platform. The code will be provided with platform processor information, to optimize for target hardware, in form of pragmas. The ordinary sequential code does not care about the memory hierarchy present, but for efficient usage of target memory hierarchy the code need to be analyzed for its memory accesses, identify extensive accesses to some small data and making it available to local memory. Optimizing code for target hardware improves overall application performance and offers efficient energy consumption. The proposed programming model is presented in Figure 3. Each part is discussed in detail in this chapter.

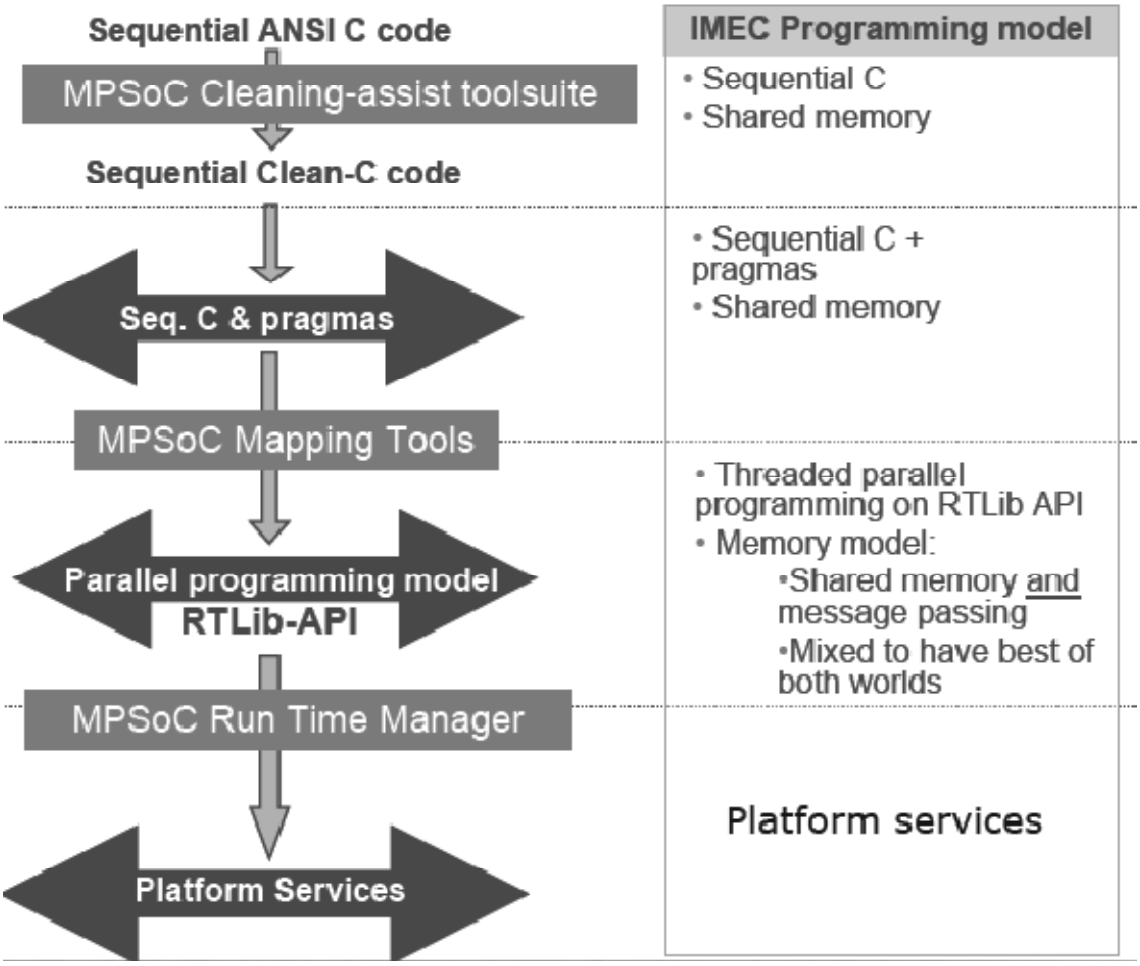

Figure 3 MPSoC Programming Model[5].

2.1 ADRES

ADRES(Architecture for Dynamically Reconfigurable Embedded System) tightly couples a VLIW processor and a coarse-grained reconfigurable matrix. ADRES provides single architecture integration for VLIW processor and coarse-grained reconfigurable matrix. This kind of integrations provides improved performance, simplified programming model, reduced communication cost and substantial resource sharing [6].

2.1.1 ADRES Architecture

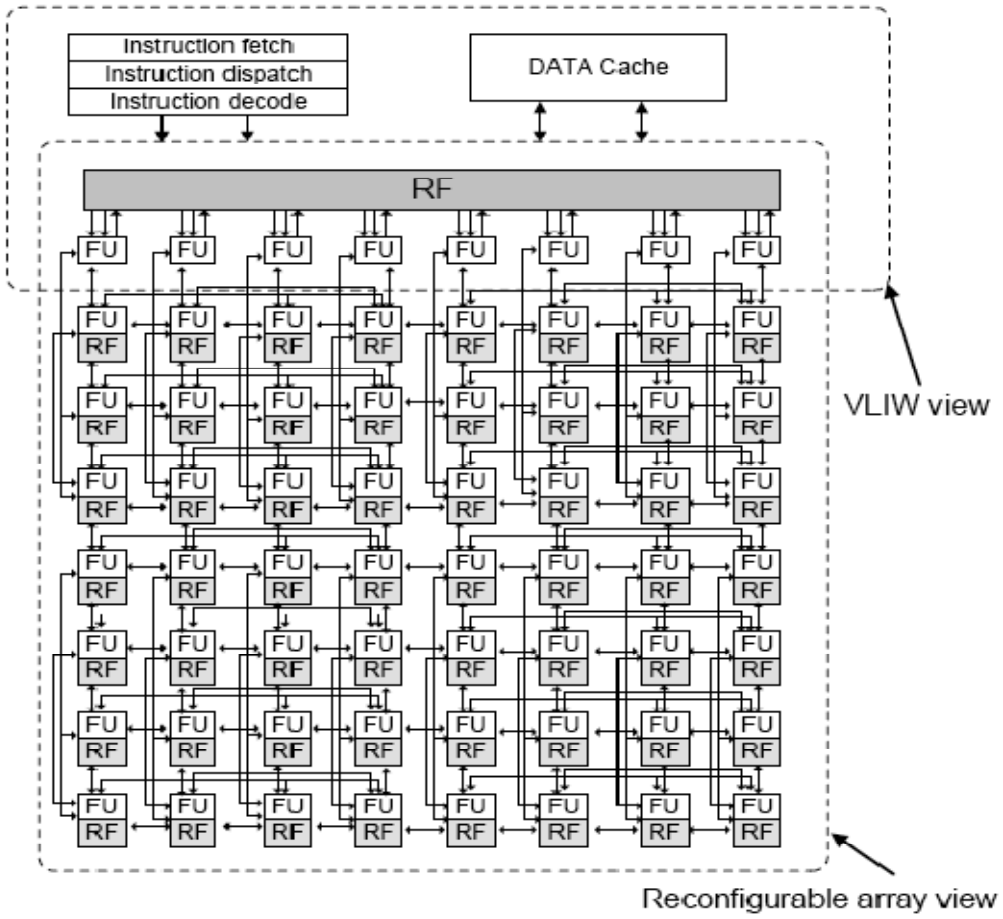

Figure 4 Architecture of the ADRES Coarse Grain Reconfigurable Array[3]

The ADRES core consists of many basic components, including mainly FUs, register files(RF) and routing resources. The top row can act as a tightly coupled VLIW processor in the same physical entity. The two parts of ADRES share same central register file and load/store units. The computation-intensive kernels, typically dataflow loops, are mapped onto the reconfigurable array be the compiler using the

module scheduling technique[7] to implement software pipelining and to exploit the highest possible parallelism, whereas the remaining code is mapped onto the VLIW processor[6-8].

The two functional views of ADRES, the VLIW processor and the reconfigurable matrix, share some resources because their executions will never overlap with each other because of processor/co-processor model[6-8].

For VLIW processor, several FUs are allocated and connected together through one multi-port register file, which is typical for VLIW architecture. These FUs are more powerful in terms of functionality and speed compared to reconfigurable matrix's FUs. Some of these FUs are connected to the memory hierarchy, depending on available ports. Thus the data access to the memory is done through the load/store operation available on those FUs[6].

For the reconfigurable matrix, there are a number of reconfigurable cells (RC) which basically comprise FUs and RFs too. The FUs can be heterogeneous supporting different operation sets. To remove the control flow inside loops, the FUs support predicted operations. The distributed RFs are small with fewer ports. The multiplexers are used to direct data from different sources. The configuration RAM stores a few configurations locally, which can be loaded on cycle-by-cycle basis. The reconfigurable matrix is used to accelerate dataflow-like kernels in a highly parallel way. The access to the memory of the matrix is also performed through the VLIW processor FUs.[6]

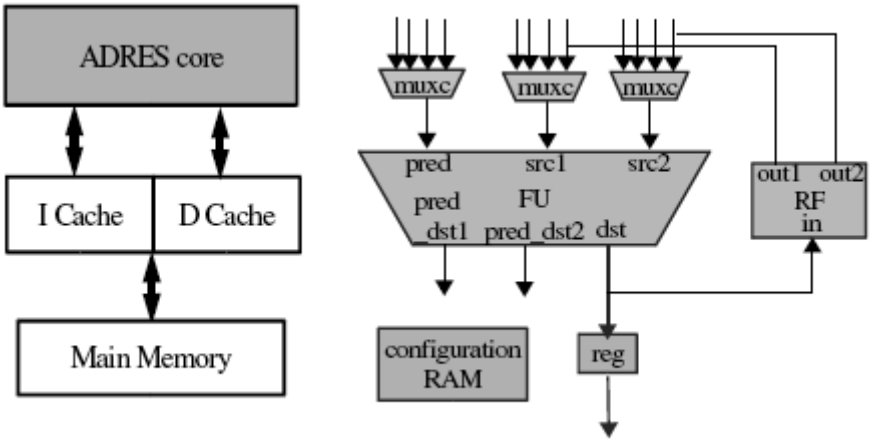

Figure 5 - ADRES system (left) Reconfigurable Cell (right)[8]

In fact, the ADRES is a template of architectures instead of a fixed architecture. An XML-based architecture description language is used to define the communication topology, supported operation set, resource allocation and timing of the target architecture[7].

2.2 Memory Hierarchy

Using a memory hierarchy boosts performance and reduces energy by storing frequently used data in small memories that have a low access latency and a low energy cost per access[9]. The easiest approach is to use hardware-controlled caches, but caches are very energy consuming so not suitable for portable multi-media devices. Moreover, cache misses may lead to performance losses[9].

In multimedia applications, access patterns are often predictable, as they typically consist of nested loops with only a part of data heavily used in the inner loops[9]. Using closer memories to processor for this heavily used data, can leverage faster and energy-efficient processing. In this case, Direct Memory access will transfer between background memories and these closer memories in parallel with computation executions.

2.2.1 Scratch Pad Memories

The software controlled scratch pad memories (SPM) can produce better performance and efficient energy consumption. Memory hierarchy utilization can be improved by design-time analysis. Scratch pad memories are difficult to implement, as analysing application and then selecting best copies and scheduling the transfers will not be easily manageable job. To handle this issue IMEC has developed a tool called Memory Hierarchy (MH)[9].

2.2.2 THE MH TOOL

The MH tool uses compile-time application knowledge and profiling data to first identify potential and beneficial data copies, and secondly to derive how the data and the copies have to be mapped onto the memory architecture for optimal energy consumption and/or performance[9].

The figure below shows the inputs and outputs of the MH tool. The platform description describes which memory partitions exist, their properties and how they are connected. Cycle count information is selected for all computational kernels by running automatically instrumented code on an instruction-set simulator. Memory access and loop iteration counts are collected separately by source-code instrumentation and subsequent execution. The data-to-memory assignment describes data-structures allocation to memory. The MH tool is responsible for data-reuse analysis and its optimization[9].

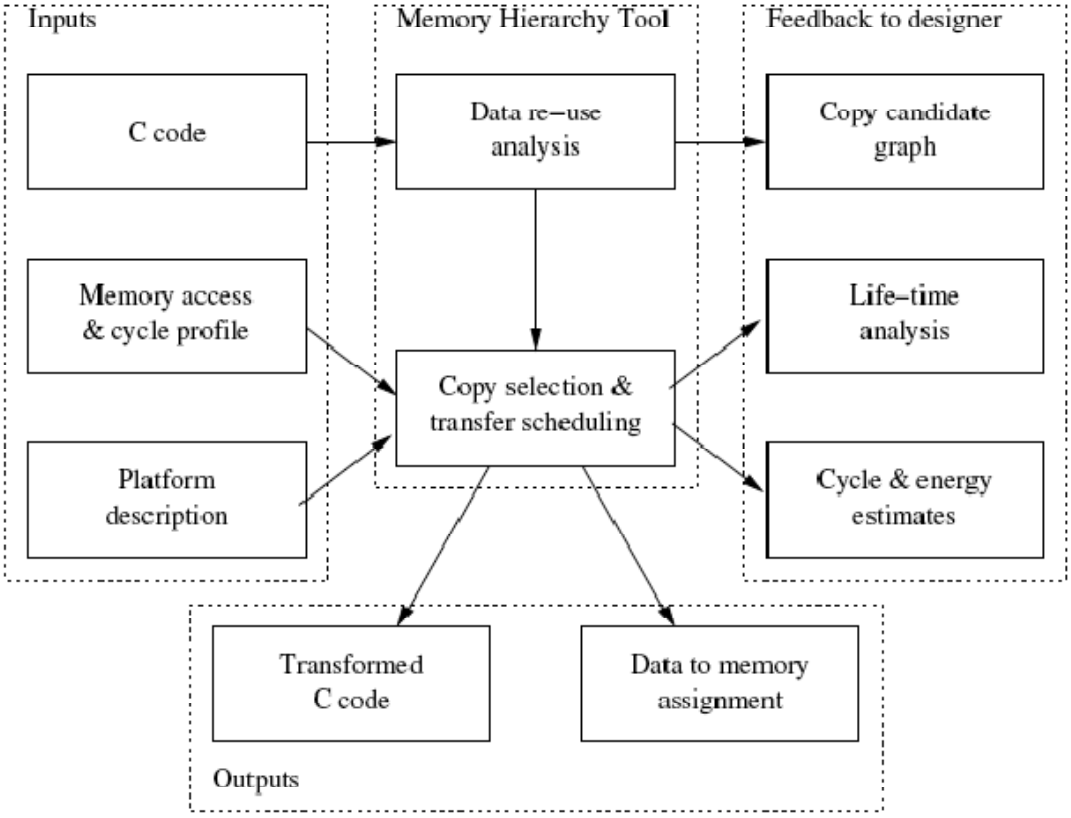

Figure 6 MH overview[9].

2.2.2.1 Data-reuse analysis

In first step code is analyzed for re-use opportunities. All global and stack arrays are considered and also a limited class of dynamically allocated arrays. For data-reuse analysis, loop boundaries are important places to explore. If in a nested loop, a small part of a large array is being accessed heavily, then it will be better to move that part to local memory to save energy[9].

2.2.2.2 Optimization

MH takes into account two cost factors: the cycle cost and the energy due to memory accesses. There may be several optimal solutions each targeting different objective. These solutions can be connected through a Pareto curve. Optimization phase may target two issues: first, how to traverse the solution space of possible assignments; the second is how to minimize the overhead of transferring data[9].

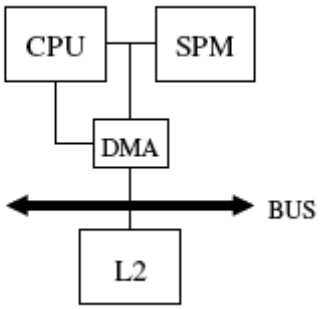

Figure 7 Scratch pad based platform architecture

The tool searches heuristically, starts by calculating the potential gain, in terms of energy, processing cycles or a combination of these, for each copy-candidate if it was assigned to the SPM (Scratch Pad Memory). The copy candidate with highest gain is then selected for assignment to the SPM. The next most promising copy candidate is selected by re-calculating the potential gains, as the previous selection may influence the potential gains of the remaining copy-candidates[9].

For MPEG-4 encoder application, the MH tool evaluates about 900 assignments per minute, of which 100 are valid[9].

For transfer scheduling, the MH tool will try to issue block transfers as soon as possible and to synchronize them as late as possible. This may expand the life-span of the copy buffers, and may arise a trade-off between buffer size and cycle cost. To minimize process stalls, a DMA unit will transfer block in parallel with the processing by the processor. Block transfer issues and synchronizations can also be pipelined across loop boundaries, to exploit even more parallelism. This kind of pipelining will also cost for more extended copy buffer life span.

2.3 The Run-time Manager

For a multiple advanced multimedia applications scenario, running in a parallel on a single embedded computing platform, where each application's respective user requirements are unknown at design time. Hence, a run-time manager is required to match all application needs with the available platform resources and services. In this kind of scenario, one application should not take full control of resources, so one need a platform arbiter. Here run-time manager will act as platform arbiter. This way multiple applications can coexist with minimal inter-application interference. The run-time manager will not just provide hardware abstraction but also give sufficient space for application-specific resource management[10].

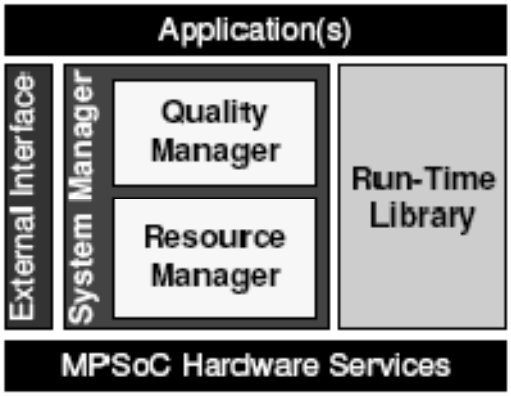

Figure 8 the run-time manager[10].

The run-time manager is located between the application and platform hardware services. Run-time manager components are explained below.

2.3.1 Quality Manager

The quality manager is a platform independent component that interacts with the application, the user and the platform-specific resource manager. The goal of the quality manager is to find the sweet spot between

The capabilities of the application, i.e. what quality levels do the application support.
The requirements of the user, i.e. what quality level provides the most value at a certain moment.
The available platform resources.

The quality manager contains two specific sub-functions: a *Quality of Experience* (QoE) manager and an *operating point selection manager[10]*.

The QoE Manager deals with quality profile management, i.e. what are, for every application. The different supported quality levels and how are they ranked[11, 12].

The operating point selection manager deals with selecting the right quality level or operating point for all active applications given the platform resource and non-functional constraints like e.g. available energy[10]. Its overall goal is to maximize the total system application value[13].

2.3.2 Resource Manager

The resource requirements of an application are well known after selecting operating point. The run-time resource manager gives flexibility in mapping task graph to MPSoC platform. Application tasks need to be assigned to processing elements, data to memory and that communication bandwidth needs to be allocated[10].

For executing the allocation decisions, the resource manager relies on its *mechanisms*. A mechanism describes a set of actions, their order and respective preconditions or trigger events. To detect trigger events, a mechanism relies on one or more monitors, while one or more actuators perform the actions[10].

2.3.3 Run-Time Library

The run-time library implements the primitives APIs used for abstracting the services provided by the hardware. These RTLib primitives are used to create an application at design-time and called by the application tasks at run-time. The run-time library also acts as interface to the system manager on different levels[10]. RTLib primitives can be categorized as under[10]:

Quality management primitives link the application or the application specific resource manager to the system-wide quality manager. This allows the application to reinitiate the selected quality.
Data management primitives are closely linked to programming model. The RTLib can also provide memory hierarchy primitives for managing a scratchpad memory or software controlled caches.
Task management primitives allow creating and destroying tasks and manage their interactions. The RTLib can also provide primitives to select a specific scheduling policy, to invoke the scheduler or to enable task migration and operating point switching.

The run-time cost of the RTLib will depend on its implementation: either in software executing on the local processor that also executes the application tasks or in a separate hardware engine next to the actual processor[10].

2.4 Clean C

The C language provides much expressiveness to designer, but unfortunately, this expressiveness makes hard to analyze C program and transform to MPSoC platform. *Clean C* provides designers with capability to avoid constructs, which are not well analyzable. The sequential *Clean C* code will be derived from sequential ANSI C. This derivation gives better mapping results[14].

Clean C gives a set of rules, can be divided into two categories as code *restrictions* and code *guidelines.* Code restrictions are constructs that are not allowed to be present in the input C code, while code guidelines describes how code should be written to achieve maximum accuracy from mapping tools[14].

If existing C code does not follow the Clean C rules, a complete rewrite is not necessary to clean it up. A code cleaning tool suit will help to identify the relevant parts for clean-up and will try to clean up most frequently occurring restrictions[14].

While developing new code, the Clean C rules can be followed immediately. But as following rules related to code structure are difficult to follow while evolving code, the code cleaning tool suit will support the application for Clean C rules during code evolution[14].

Some of the proposed guidelines and restrictions are as under: details can be found in [15].

- Overall code architecture
    - Restriction: Distinguish source files from header files
    - Guideline: Protect header files against recursive inclusion
    - Restriction: Use preprocessor macros only for constants and conditional exclusions
    - Guideline: Don’t use same name for two different things
    - Guideline: Keep variable local
- Data Structures
    - Guideline: Use multidimensional indexing of arrays
    - Guideline: Avoid **struct** and **union**
    - Guideline: Make sure a pointer should point to only one data set
- Functions
    - Guideline: Specialize functions to their context
    - Guideline: Inline function to enable global optimization
    - Guideline: Use a loop for repetition
    - Restriction: Do not use recursive function calls
    - Guideline: Use **switch** instead of function pointer

2.5 ATOMIUM / ANALYSIS (ANL)

Multimedia applications are mostly data dominant. To realize efficiency of such applications data transfer and storage is crucial to explore[16]. For a complex multimedia application like MPEG-4 encoder this high level analysis is essential input for efficient hardware/software partitioning[17].

The huge C code complexity of multimedia applications makes a complexity analysis tedious and error-prone. Atomium/Analysis gives additional support to handle such complexities. This tool consists of scalable set of kernels and tools providing functionality for advanced data transfer and storage analysis and pruning[17].

Atomium/Analysis can be mainly used for three purposes[18].

Analyze memory usage of array data in data dominated C programs and to identify the memory related bottlenecks.

Prune unused parts of C code based on profiling information.

Accurate cycle count information for parts of C code, called ‘kernels’.

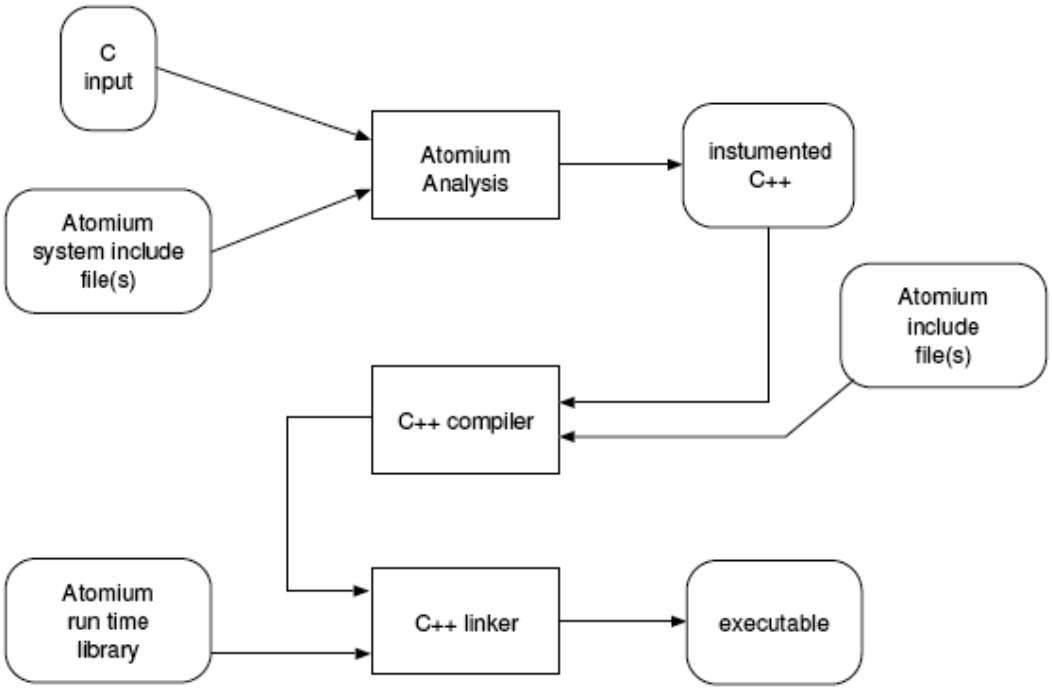

Figure 9 Atomium/Analysis Instrumentation Process[18]

2.5.1 Atomium/Analysis Process Flow

Atomium/Analysis tool will take C files as input, and then it will insert some C preprocessor macros into the code and write code again in C++ style. As the output is C++ files, so we need C++ compiler and will be linked with ATOMIUM run time library.

When run, besides performing its normal functionalities, generate access statistics for the arrays present in the code[18].

Atomium/Analysis on given C program involves six steps[18]:

1. Prepare program for ATOMIUM.
2. Instrument the program using Atomium/Analysis.
3. Compile and link instrumented code.
4. Generate instrumentation results.
5. Generate an access count report.
6. Interpret the instrumentation results.

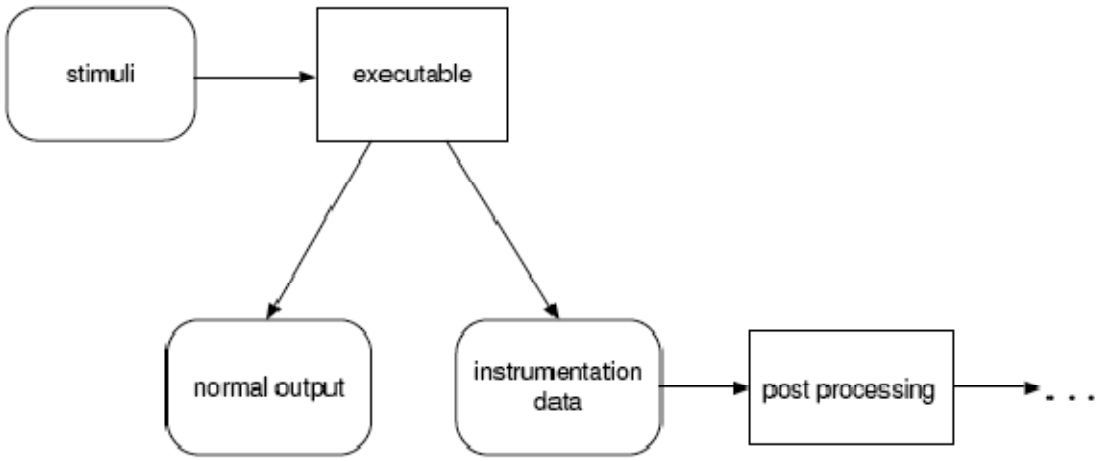

Figure 10 Atomium/Analysis Generating Results[18]

Atomium/Analysis will be explained stepwise *Methodology and Implementation* chapter.

2.6 MPA TOOL

The MPA tool assists the user in parallelizing sequential code for running it on multi processor platform. The general idea of the MPA tool is that the user identifies the parts of sequential code that are heavily executed and should be executed by multiple parallel threads to improve the performance of the application. The pieces of the code that will be parallelized are called parsections[19].

For each parsection, the user specifies how many threads must execute it and what part of each of these threads must execute. The user can divide the workload in terms of functionalities, loop iterations, or both depending on what is most appropriate for given parsection[19].

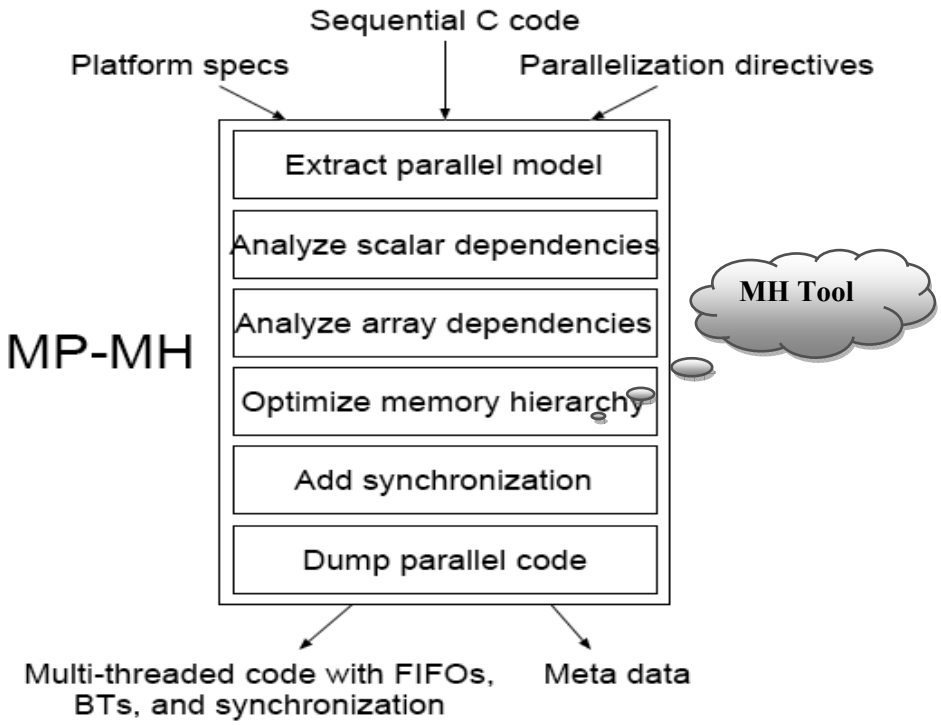

Figure 11 MP-MH Tool Flow[19].

These parallelization directives have to be written in a file provided by Atomium/MPA tool. The main reason for using directives instead of pragmas inserted in the input code, is that it simplifies exploration of multiple parallelizations for the same sequential code[19].

Given the input code and parallelization directives, the tool will generate a parallel version of the code and insert FIFOs and synchronization mechanisms where needed.

The tool settles the dependencies between threads. However, designer has optional control over shared and private variables. By specifying a variable shared or private, the designer will take care of its communication and synchronization mechanism.

The generated parallelized code must be compiled and linked with MPSoC Run Time Library (RTLib). The user may go for high-level functional simulation of parallelized code or may compile code for a real MPSoC platform.

Finally, when the parallelized code is compiled and run, it can dump an *act.vcd* file containing traces of the parallelization run in "Value Change Dump" format. This feature requires that the input code use Atomium Record Playback (ARP) functions.

In addition to provide parallelization, The MP-MH tool will also optimize the storage and data in the hierarchical distributed memory architecture of the multiprocessor platform. It will decide where and how each array must be stored in the memory hierarchy, which local copies have to make to increase efficiency, where these local copies must be stored, and when they must be made.

2.7 High Level Simulator (HLSIM)

When combined with Atomium Record & Playback functionality, the parallelized code linked with the functional version of the MPSoC RTLib can be used as a high-level simulator for the parallel code. This allows quickly getting an idea how well the parallelization will perform. The user will supply some platform parameters to the high-level simulator to simulate the delays introduced by the communication and synchronization functionality[19].

The HLSIM reads the platform architecture as input file, which describes physical resources available on platform, such as processors, memories, DMAs. Extra information regarding memories and DMAs is provided e.g. read/write ports, size of memory, page size. Such kind of information is used by HLSIM while simulating application[20].

Currently, only FIFO delay parameters are implemented, but in the future also thread spawning/joining, and block transfer delay will be added.

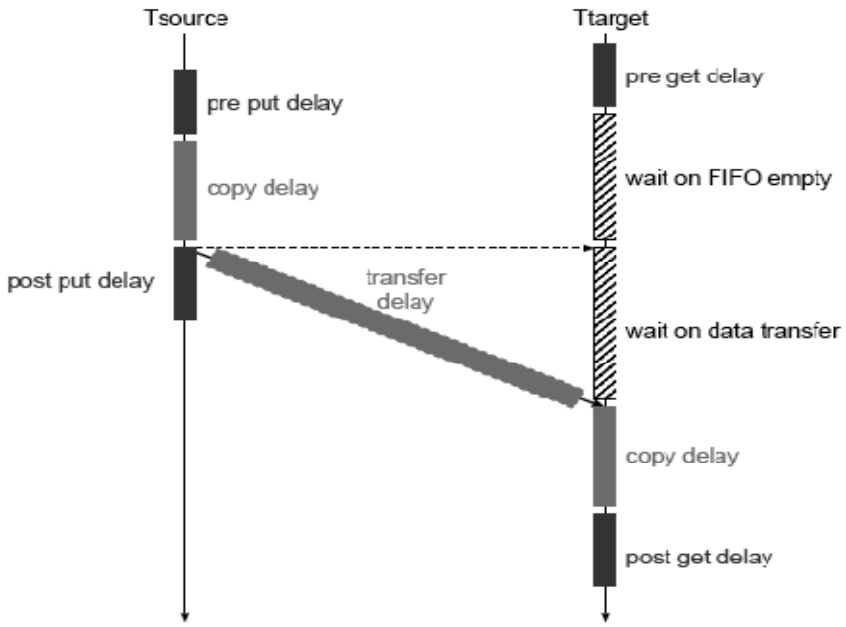

Figure 12 Illustration of FIFO delay[19].

The delays shown in blue are fixed delays while in read are proportional to FIFO element size. The black hashed line indicates that thread has to wait for something. If FIFO is full when the put is issued, the source thread will have to wait until a FIFO slot becomes available again[19].

Typical multimedia application code will contain nested loops for which most of processing time is spent. These nested loops are called *kernels*. The Atomium Record & Playback introduce timing information for these kernels, which are important timing-accurate running[20].

Atomium analysis tool [18] profiles the application code and inserts ARP instrumentation calls to add the timing information around identified kernels in the application code. The exact execution time information for these kernels is then obtained by running these kernels on a stand-alone Instruction Set Simulator for the processor used in the platform[20].

HLSIM first reads information about the platform architecture. With this information, it becomes aware of platform resources. When application starts configuring RTLib component information, HLSIM accumulates that information with already known platform information. ARP instrumentation information and BT information are also read and accumulated to be accessed during simulation run[20].

RTLib API uses FIFOs and/or BTs to do thread communication and uses loopsyncs to do thread synchronization. Currently BTs are modeled as transfer over a connection between two platform nodes whereas FIFOs and loopsyncs are modeled in HLSIM using shared memory[20].

Chapter 3 Previous Work

The MPEG-4 encoder is one of resource hungry but commonly used multimedia applications for handheld devices. The application consists of a set of kernels, and each kernel consumes and produces a large amount of data to be communicated between different memory hierarchy levels. Furthermore, application is computationally extensive, thus the distribution of the kernels across several processing elements is wishful[21]. Thus, MPEG-4 encoder is potential application to be explored. The sequential MPEG-4 encoder's implementation is processed through previously explained programming model and tools, to generate several parallelization possibilities.

3.1 Experimental workflow and code preparation

The experimental workflow is depicted in figure. The input code must be clean according to the *Clean C* rules (on page 12). The MH tool (on page 7) introduces copies and Block Transfers (**BT**) in the code. Minor manual modifications are needed to prepare this code such that MPA (on page 15) can produce correct output code. This code is executable on the High Level Simulator (on page 17) [22].

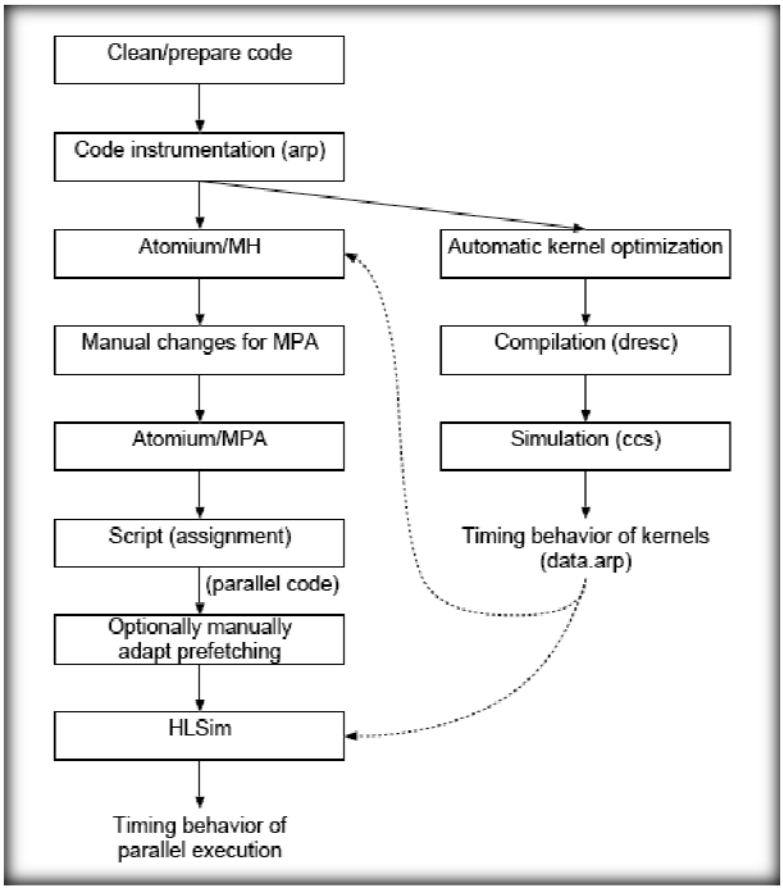

Figure 13 Experimental tool flow[22].

The MH and MPA have been modified to make it easier to generate a consistent code[23].

3.2 MPEG-4 Encoder Experiment

The experiments are performed for the "crew" sequence at 4CIF resolution. The sequence is first profiled with Atomium Record Playback (**ARP**) profiling data. From these profiling results, we may conclude a rough decision about how load is distributed between functionalities inside application. The MPA (on page 15) and MH (on page 7) refinement and execution on the HLSIM (on page 17) enables to quickly iterate over these rough directions and get sensible feedback[22]. This kind of iterations leads to several solutions for different performance requirements.

The top-level cycle distribution over the various functionalities inside mpeg-4 encoder is shown in Figure 14.

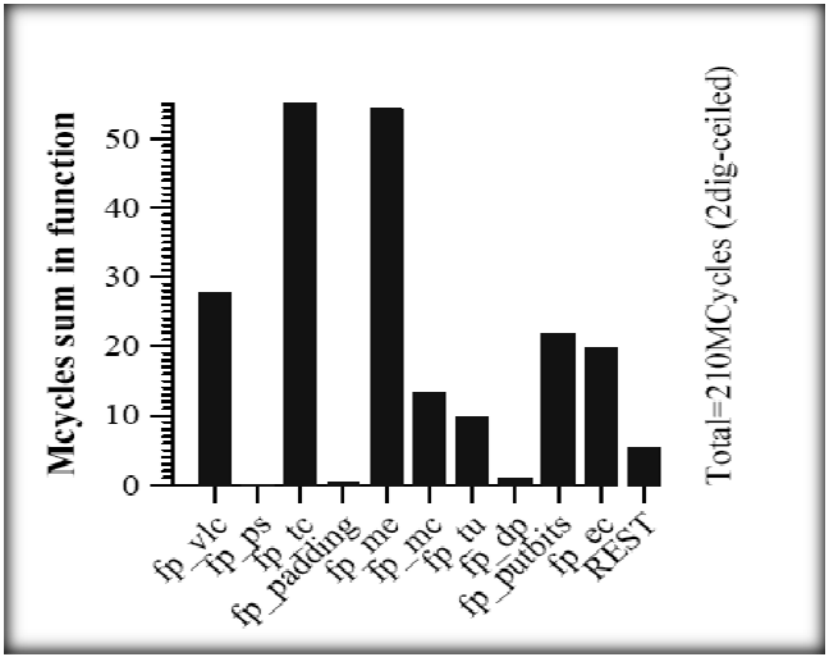

Figure 14 Top-level cycle distribution for 10 frame of crew 4CIF sequence[22].

The **motion estimation** and **texture coding** are equally dominant and form the majority of all cycles. Based on the cycle count distribution over the different functions, a rough decision can be deduced regarding parallelization options for balanced processor load. Some functionalities here are highly sequential in nature, so cannot be parallelized at data level. Therefore, it will be better to keep them sequential and combined with other functionalities, making small contributions, which are not worth to make their own thread. This kind of sequential nature functionalities limits the maximum performance gain to a factor of 6.

Figure 15 MPEG-4 Encoder parallelization options for various processor count[22].

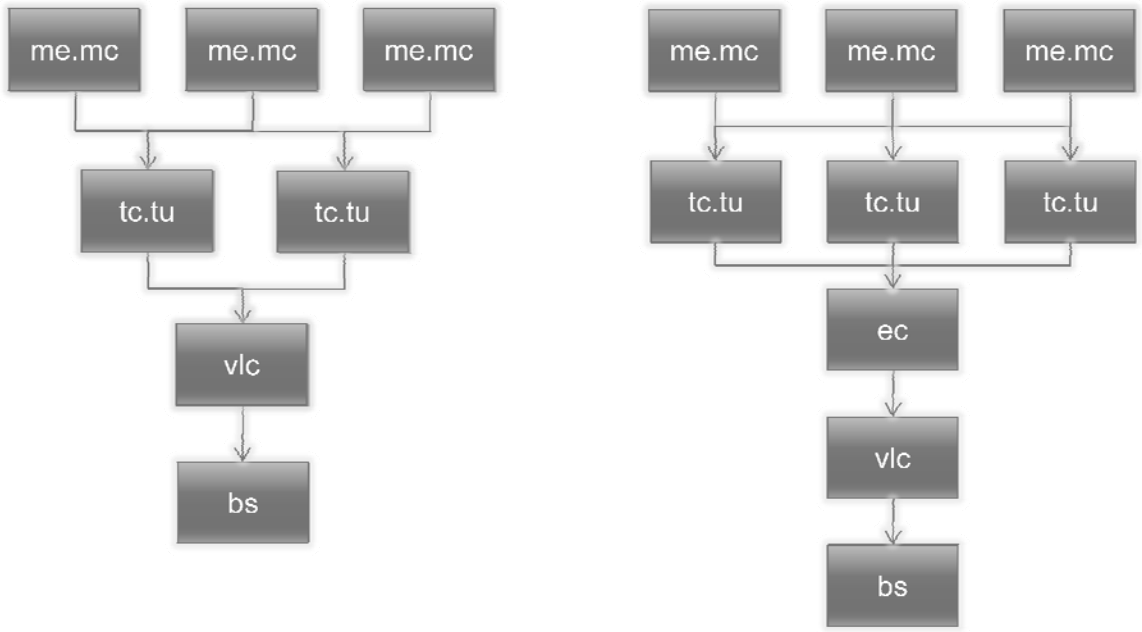

Figure 16 MPEG-4 Encoder parallelization options for various processor count[22].

A number of parallelization options were derived, keeping in mind maximum speedup factor, ranging from 1 processor to 9 processor. The 3 threaded parallelization (**me.mc→tc.tu→vlc.bs**) is a pretty balanced functional pipeline of three stages, and gives a noticeable performance gain over sequential one. A bit more gain can be achieved by further pipelining **me** and **mc**. Furthermore, we may exploit the coarse data parallelism. A data split for the motion estimation is complicated due to dependencies in the motion vector prediction. First half of first **MB** line should be processed first, and then second half of first line and first half of second line can be processed in parallel.

3.3 Platform Template

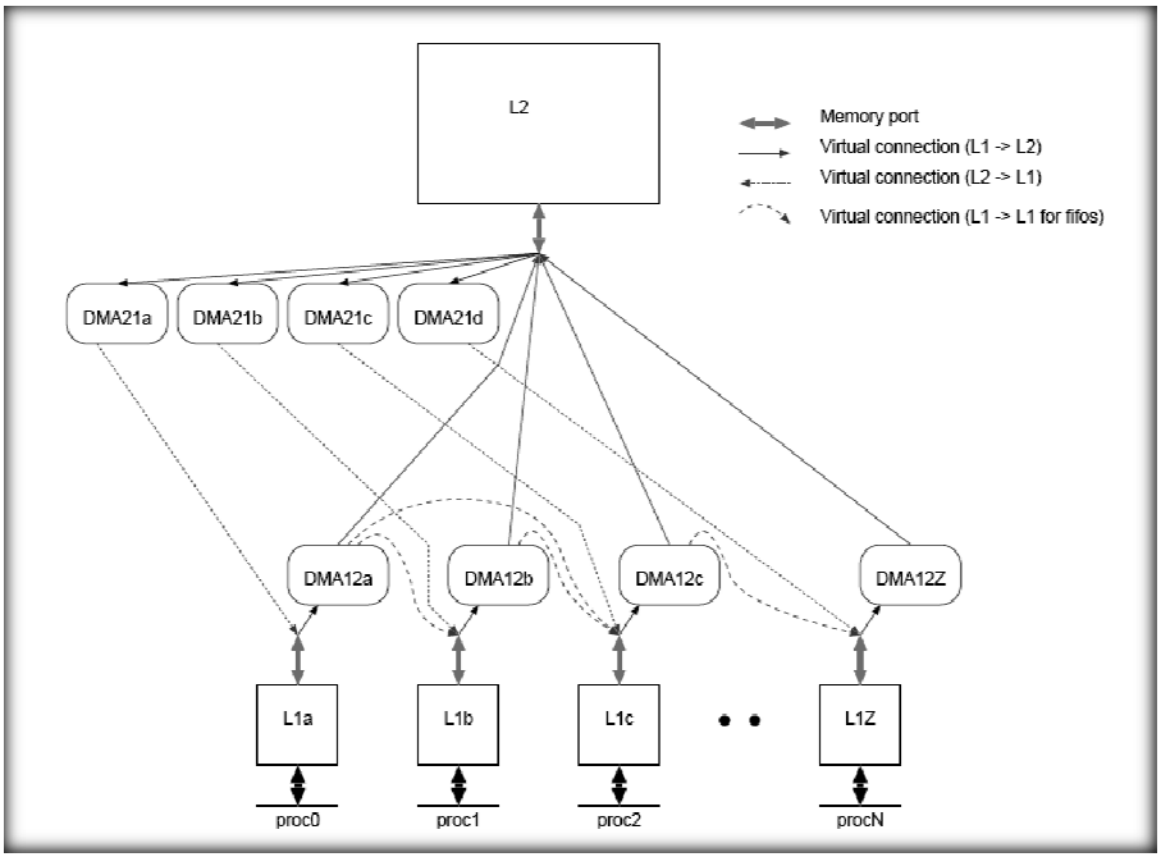

Figure 17 Platform Instantiation[22].

The delays of the system are always expressed relative to one processor cycle. All processors have currently same speed. Each processor has a local L1 memory and there is one shared L2 memory. Each L1 has a local DMA. The DMA reads from the local memory and can write via a connection to remote memory[22].

The virtual connections are not a platform characteristic but an application implementation detail. Therefore, these connections are setup inside application. These connections are unidirectional.

Moreover, L1→L1 connections are FIFO communication, must be step up by application too. At L2 side, separate DMAs are instantiated for each L1 memory to keep a fast response. If only one DMA deal with all BT requests from L1 memories, a small BT from one thread can be blocked for a long time by a large BT from one of the other threads. Instantiating a DMA per L2→L1 connection provide more predictability and improves speedup[22].

Chapter 4 Methodology & Implementation

The parallelizations proposed in [22] have used Atomium Record & Playback profiling to obtain simulation cycles. The parallelizations are derived based on the criterion of possible maximum performance gain. That work has a number of issues that have been targeted by this chapter.

First, we need to profile sequential code with respect to memory accesses to have insight regarding memory bandwidth usage by the application. This information will be useful to analyze and support previous work and propose new parallelization possibilities.

Second, proposed parallelization solutions need to be profiled and criticized. This will be helpful to understand the application's behavior when it is pipelined with multi threads.

Third, until now we may decide our parallelization at compile time. Therefore, once application is allocated hardware resources, a designer has no control to re-configure the resources of CPU/memory. Our target is to delay a designer's decision until run-time. A successful implementation will provide a designer with the run-time switching from one parallelization to another. Briefly, one will be able to set performance/cost parameters at run-time and may allocate and de-allocate hardware resources at run-time.

4.1 ANL Profiling for Sequential MPEG-4 Encoder

Atomium Analysis Tool instruments data-dominated C programs with the purpose of obtaining various kinds of profiling information. ANL [18] can be used to access counting, dynamic pruning and kernel profiling. Here our target is to profile for access count, as we are interested in array accesses by the MPEG-4 encoder.

ANL inserts some C-preprocessor macros into the original code, which enables the program to generate access statistics for the arrays used in the code.

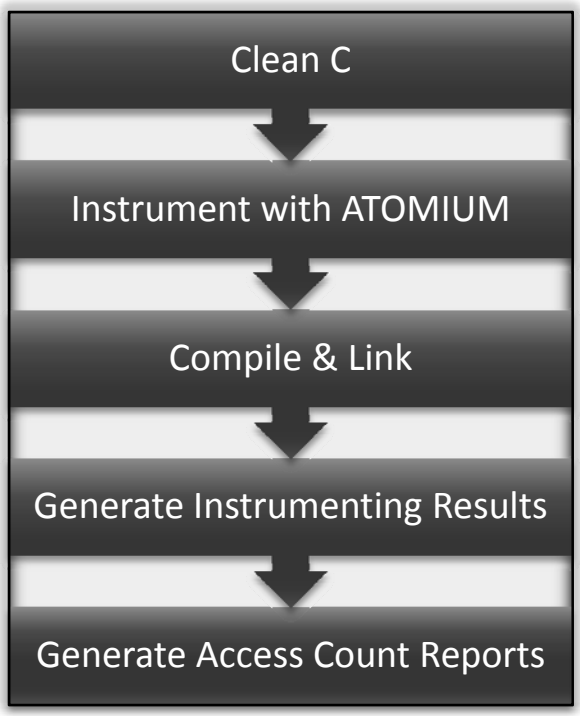

Figure 18 Profiling Steps for ANL.

ANL profiling involves following steps:

4.1.1 Preparing the program for ATOMIUM

First, the program should adapt ATOMIUM C restrictions (on page 12). ATOMIUM comes with a pair of auxiliary pre- and post processing scripts to analyze code of Clean C restrictions. For current scenario, pre processing will be enough to the job. The pre-processing script named **preato** will generate the new code in a new directory **atoready**. The **preato** will combine all header files to one header file **AllInOne.h**.

The newly generated cleaned code will be used in next step for Instrumenting with ANL.

4.1.2 Instrumenting the program using ATOMIUM

Instrumenting will be done by invoking the appropriate ANL command. We could instruct the tool to instrument the .c files found in the **src** directory for detailed local access statistics and putting the instrumented code in **outdir** directory.

4.1.3 Compiling and Linking the Instrumented program

The instrumentation process will result in instrumented code dumped in **outdir**. The directory **outdir** will contain ANL header files and **src** directory contains the instrumented code. ANL takes C code as input and output instrumented is in C++ format (Figure 9). For compilation, we will use C++ compiler and link it with ATOMIUM runtime library.

After successful compilation and linking, executable file for application will be created. Running this executable will generate access statistics beside its normal output (Figure 10). Beside producing normal output, a new file will be generated named **ato.data** by default.

In the instrumented program, data may end up in other places in memory than is the case in the original un-instrumented one. The buggy program that happen to work by accident, may fail unexpectedly when instrumented. The RTLib (on page 11) sometimes detect such errors e.g. array being accessed outside its boundary. Still such accesses will be reported in report[18].

4.1.4 Generating an Access count report

The Atomium data file can generate a variety of reports. The generated report can be in form of HTML, TABLE or TREE format. The HTML format gives better navigability but requires a lot of hard disk space.

The report may include array access count, peak memory usage and function & block activation. For current application, we will go through all three kinds of reports to explore application in depth and have comprehensive profiling results.

4.1.5 Interpreting results

The reports generated for current application is huge containing 145 functions/blocks accessing 73 arrays. Therefore, we need some criteria here. We can either define the criteria as the arrays being accessed most or functions whose execution time is longer. In first criteria, we can identify arrays but tracking location inside application for these accesses is bit tricky. For second criteria, if we take function execution time as starting point, we can easily identify arrays being accessed by these functions. We can also notice that function's execution time is higher because of heavy array accesses or intense functional processing. By

identifying such functions, we can analyze the program behavior and can investigate parallelization possibilities.

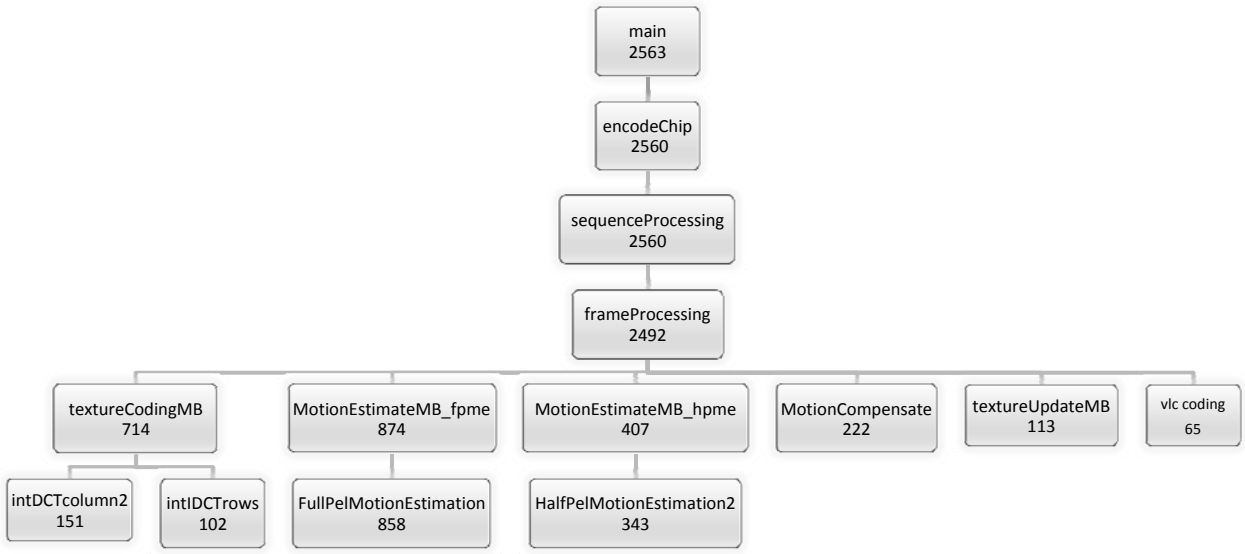

Figure 19 Execution Time Distribution for MPEG-4 Encoder (in microseconds)

The hierarchical view of execution time shows that mainly execution is done by **frameprocessing** and its sub-functions. The time shown for each function is sum of its own execution time and its sub-function's execution time. We further explore in hierarchical depth to find functions mainly contributing in execution time e.g. **MotionEstimateMB_fpme** having execution time as 874 microseconds but its sub-function **FullPelMotionEstimation** takes 858 microseconds, so parent function take very less time for self execution.

After identifying these functionalities, we will go in detail for arrays being accessed by these selected functionalities and how much accesses are being covered inside this small group of functionalities compared to total accesses by application.

Access count statistics for these functionalities are explored and explained one by one in next chapter (**Error! Reference source not found.**).

4.2 Discussion on Previous Parallelizations

Several parallelization possibilities were proposed for MPEG-4 encoder on basis of **ARP** profiling (Figure 15, Figure 16). Each parallelization possibility is intended for a particular cost-performance criterion and a specific processor count. First step here would be to profile each of these parallelizations and evaluate their performances. As in next step, we will try to switch between two parallelizations at runtime. Therefore, this profiling will be useful to select optimal parallelizations, i.e. parallelizations that give us considerable gain in form of performance and memory bandwidth usage.

Profiling will be done one by one for each parallelization possibility. MPA (on page 15) will be directed to generate each parallelization following **MP-MH** tool flow (Figure 11). Dumped parallelization code will be run on **High Level Simulator** (on page 17).

The **High Level Simulator** output will be profiling results for current simulated parallelization. Profiling results are in very much detail about each platform component. This report gives detailed information for each thread simulation cycles, FIFO data transferred and physical memory accesses & data transferred for each thread.

Results are explored thoroughly for each parallelization and calculated for total performance and memory accesses.

The Figure 20 shows total simulation cycles and memory accesses (FIFO & L2 transactions) for all parallelization entries. The first entry named **1thread** is sequential execution as there is only thread. The "_" separates one thread's functionality from other. In general, we can see that as the number of processors involved increases, number of simulation cycles decreases and memory bandwidth utilization increases.

We can see that shifting from sequential execution to **3** threaded parallelization, **memc_tctu_vlcbs**, gives 50 percent reduction in simulation cycles consumption. While using **5** threaded parallelization, **2memc_tctu_vlc_bs**, gives 63 percent performance gain. In the same way, increasing involved processor count to **7**, **3memc_2tctu_vlc_bs**, saves 70 percent of simulation cycles.

The parallelization **me_mc_tctu_vlcbs** and **me_fpmemc_2tctu_vlc_bs** show deviation from this generalization. The **me_mc_tctu_vlcbs** is **4** threaded parallelization, but gives just 1million cycle advantage over **3** threaded parallelization **memc_tctu_vlcbs** but memory utilization is around 50 million accesses more. Similarly, **me_fpmemc_2tctu_vlc_bs** is **6** threaded parallelization but behaves much verse than **2memc_tctu_vlc_bs** (**5** threaded). These sub-optimal parallelizations entries are marked with red circles.

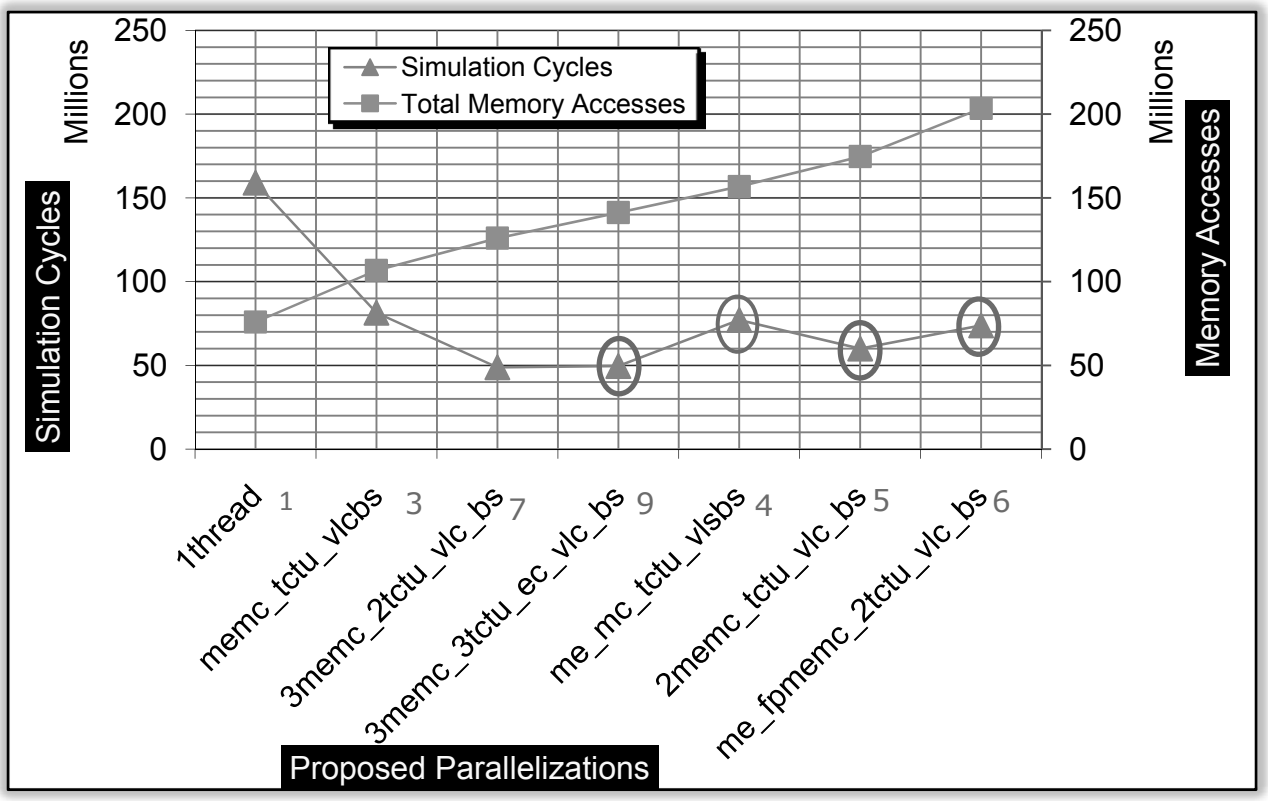

Figure 20 Profiling results for proposed Parallelizations (crew 4CIF 10 frames sequence).

A close analysis shows that separating **mc** functionality from **me** can be a reason for this deviation. The code generated from MPA tool shows that as we separate **mc** functionality from **me** in separate thread, platform has to duplicate some memory blocks from L2 to each thread processor that causes enormous increase in memory bandwidth utilization.

This critical analysis shows that decision regarding keeping **me** and **mc** functionalities in separate threads to improve performance isn't a good solution and must be avoided as it causes huge memory bandwidth utilization and gives an insignificant performance advantage.

4.3 Choosing optimal parallelizations for run-time switching

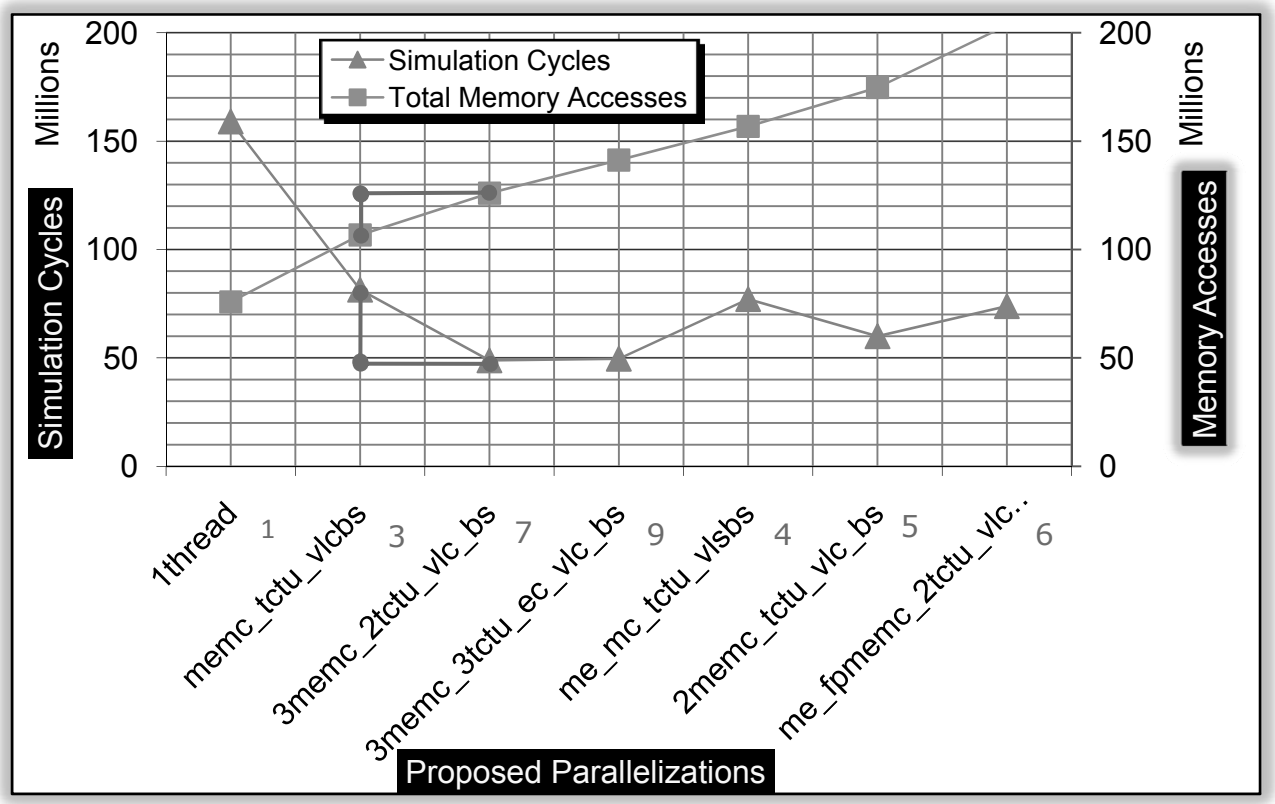

Figure 21 Optimal Parallelization Selection.

Next experiment step is to enable run-time application's switching from one parallelization to another, to have control over platform resources allocation & de-allocation and to adjust performance-cost criteria at run-time. For this experiment, we need to select two potential parallelizations, which can provide us considerable switching advantage.

The selection criteria involve two main aspects. First, number of processors involved in both parallelizations should be so distinct that we can experience benefit in terms of spare processors while shifting from greater processor count to lesser one. Second, cycle utilization for both parallelizations should be so different that we can experience performance gain while shifting from lesser processor count to greater one and can explore run-time switching advantages in best way.

Looking at the profiling results closely, shows that **3** threaded parallelization (**memc_tctu_vlcbs**) and **7** threaded parallelization (**3memc_2tctu_vlc_bs**) suits best to this criteria. We have a considerable difference in terms of processor count on one hand and secondly we will be able to explore 30 million simulation cycle and 20 million memory bandwidth accesses difference between two parallelization.

The HLSIM while reporting other profiling details for each parallelization also reports performance speedup factor in comparison with sequential execution. For **3** threaded parallelization speedup factor is reported as 2.37x and for **7** threaded parallelization it is 4.39x. These factors provide minimum and maximum performance gain from the resultant switching component based on these two parallelizations.

4.4 Run-time parallelization switcher

Now after selecting potential parallelizations, we can proceed to next step i.e. embedding two parallelizations in one component and then enabling switching between them at run-time. The new component will consist of threads belonging to both parallelizations, at one time only one set of thread will be active, and other will be inactive. One point to be noted here is that as our case study application is MPEG-4 encoder, so our switching decision will be made at start of each frame. In other words, we are experimenting inter-frame switching. Inter-frame switching is decided to keep the experiment simple and focus on exploiting tools capability.

A limitation introduced by RTLib is that execution of individual threads in a Parsection is unconditional, i.e. all threads in a parsection are spawned collectively. Our target here is to spawn threads that belong to current selected parallelization and not to overload with idle threads belongs to inactive parallelization. Therefore, these two parallelizations will lie in separate parsections and at one time only parsection will be active.

Another limitation by RTLib is that only master thread can spawn one or more slave threads, but a slave thread cannot spawn other slave threads. So our switching mechanism will be monitored by master thread. At start of each frame, master thread will activate the desired parallelization and after encoding current frame, control will be given back to master thread. Master thread will deactivate all initiated threads, wait for next frame input, and follow the same procedure.

4.4.1 Configuring new component

We need to configure new component consisting of both parallelizations. We will use the same platform template as used in previous work (on page 23). Our platform template consists of virtual memories corresponding to physical memories, virtual connections established to L2→L1 transfers and FIFO transfers, and virtual processors corresponding to physical processors.

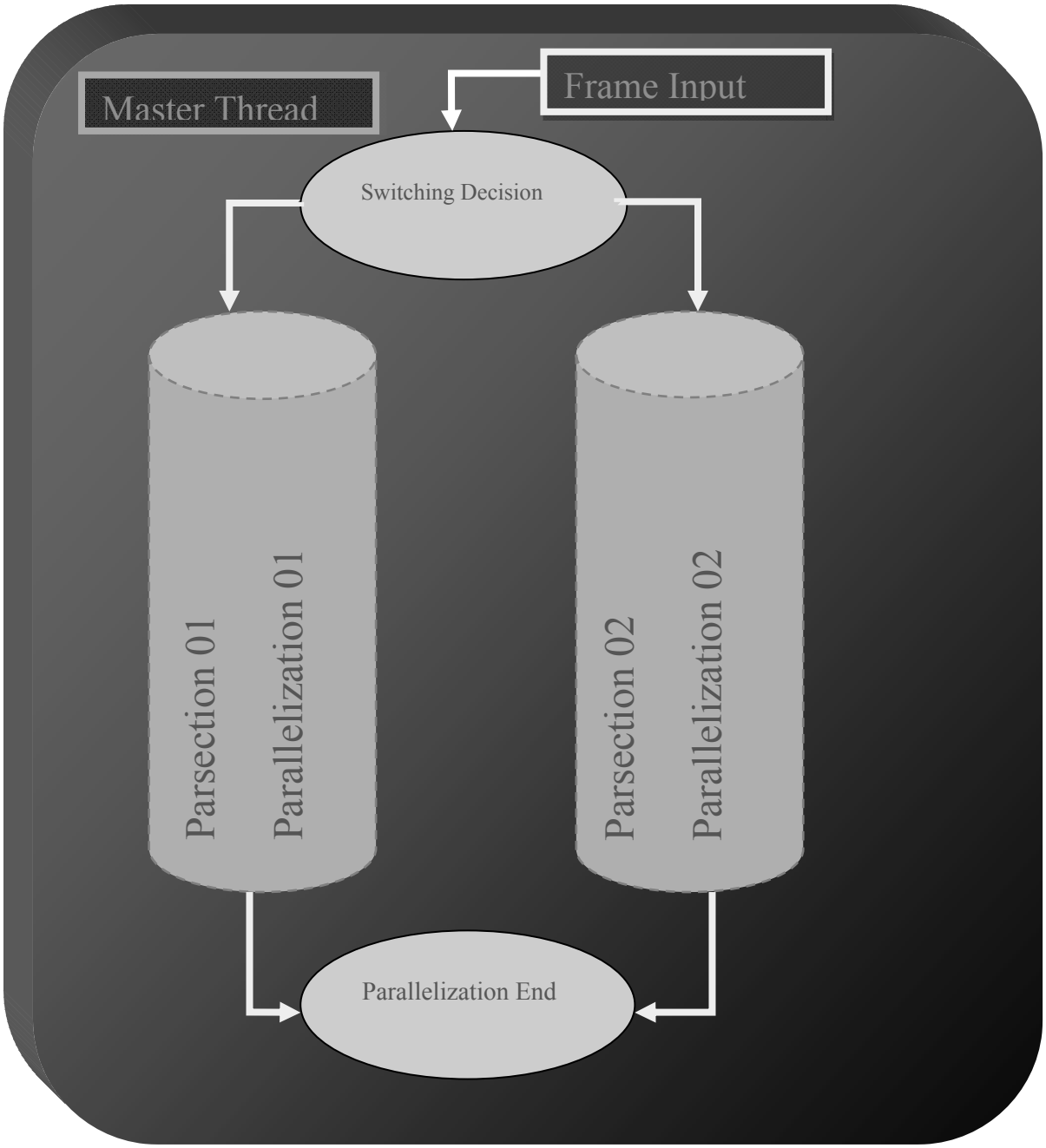

Figure 22 Parallelizations integration overview.

Figure 23 Component with both parallelizations.

For newer integrated component, we need to configure all of these elements according to our new platform structure. One parallelization **memc_tctu_vlcbs** consists of three slave threads and other parallelization **3memc_2tctu_vlc_bs** consists of seven slave threads. There will be one master thread controlling these slave threads. So in total newer component will consists of 11 threads, each corresponding to a physical processor. At one time, only 4 of them or 8 of them will be active. For virtual memories, there will be a virtual memory corresponding to a physical memory for each thread and one virtual memory will be shared memory corresponding to L2 memory. Each L1 virtual memory will be having communication to L2 through this shared memory. Communication channel is established through virtual connections, which are unidirectional. Multiple FIFOs can be established on same virtual connection, if source and destination nodes are same. Moreover, the last thing to configure is **loop_sync**, which will be configured for both parallelizations.

Each of these components: virtual memories, virtual connections, virtual processors, DMAs are addressed through their respective IDs. Each thread belongs to exactly one parsection, and all threads in a parsection will be activated at once.

The component configuration is static and it is the first job to do at when application is run. The newer component will include all FIFOs and virtual connections configured for both parallelizations previously, but here we need to adjust them according to current platform component IDs. Each FIFO & virtual connection personal IDs, source IDs and destination IDs will be properly adjusted to keep parallelization equivalent to individual parent parallelization.

4.4.2 Switching decision spot

As it has been decided that switching, decision will be made by master thread, so now we need to decide where we should take this decision for best response. A close analysis of code and application control flow shows that all slave threads are spawned inside **frameProcessing** function.

As current implementation is targeting inter-frame parallelization switching, so **sequencProcessing** is best place for switching decision. The **sequencProcessing** choose parallelization by a variable name **sel_para**. The input parameters for **frameProcessing** are also edited and a new parameter i.e. **sel_para** is added.

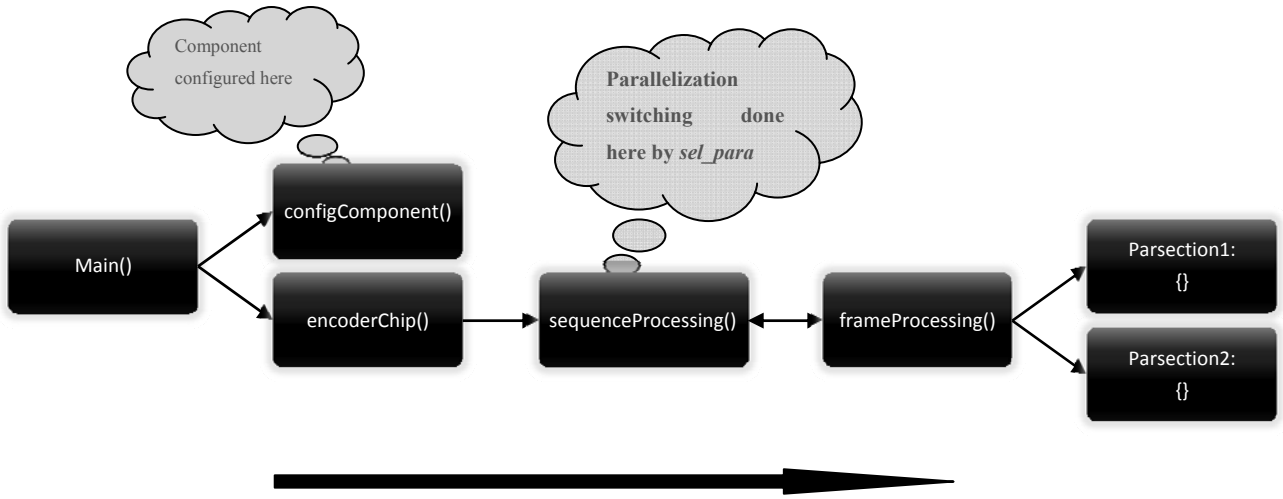

Figure 24 MPEG-4 Encoder application control flow.

```
if (sel_para == 0)
{
            Parallel_Section_01:
      {
      // Master will spawn all thread in this parallel section and
      // pass arguments to each thread and then activate threads
      // after encoding current frame, threads will be deactivated
      by // Master thread
      }
}
else if (sel_para == 1)
{
            Parallel_Section_02:
      {
      // Master will spawn all threads in this parallel section and
      // pass arguments to each thread and then activate threads
      // after encoding current frame, threads will be deactivated
      by // Master thread
      }
}
```

Figure 25 Parallelization spawning model

4.4.3 Manual adjustment to code and tools

Our programming model and tool support ends till generation of parallelization codes, dumping code and simulating it on HLSIM. The experiment regarding embedding two separate parallelizations into one component is completely handled through manual approach. Automatic embedding the two parallelizations can be a next step after successful experiment results. At this stage, we need to embed both parallelizations codes manually and adjust the previously generated codes according to newer component. This manual setup contains many different aspects to cover and will be covered one by one.

We will dump both parallelizations codes, then take one parallelization code as basic, and start adding second parallelization code into first.

4.4.3.1 Adding new threads and naming convention

The C file **frameProcessing.c** contains definition for all threads. Previously, we were having just one parallelization so thread's function naming convention was like

```
T1_memcfunctionality(), T2_tctufunctionality(), T3_vlcbsfunctionality()
```

Now we need to accommodate two parallelizations, so a new naming convention informing both about thread ID and parallelization ID.

```
P1_T1_memcfunctionality(), P1_T2_tctufunctionality(),
P1_T3_vlcbsfunctionality()
```

Similarly for second parallelization,

```
P2_T1_memcfunctionality(),P2_T2_memcfunctionality(),P2_T3_memcfunctionality()
P2_T4_tctufunctionality(), P2_T5_tctufunctionality(),
P2_T6_vlcfunctionality(), P2_T3_bitstreamFunctionality()
```

First step is to dump all threads definition in **frameProcessing.c** and then to apply new naming convention.

4.4.3.2 Parsection switching & parallelization spawning

As discussed before, for each video frame slave threads are spawned in **frameProcessing** functionality. We need to add spawning mechanism of second parallelization thread in a *switch* or *if-else* statement. Code will be following previously discussed program model (Figure 25). The parsection 02 will spawn threads, pass argument and activate threads belonging to second parallelization.

4.4.3.3 Replicating and Adjusting functionalities called by threads

MPA generated code has a naming convention for functionalities directly called by thread functions or related to a single specific thread e.g. functions called by or related to **T1_memcfunctionality** will be having prefix "T1_".

For new component, as both parallelizations have same functionality and access same functions form different thread, so we need to replicate all these functionalities and introduce same naming convention as used for newer threads i.e. "T1_" will be replaced by "P1_T1" or "P2_T1".

These functionalities mainly lie in:

- motionCompensation.c
- textureCoding.c
- textureUpdate.c
- entropyCoding.c
- textureCoeffCoding.c
- bitstreamPacketsizing.c
- motionCompensation_sep.c

Functionalities from these files will be simply copied to directed files and will be renamed according to new naming convention.

A reason to this replication is ARP instrumentation. ARP instrument each functionality and kernels inside with **arp_enter()** and **arp_leave()**, to keep track of execution path i.e. from where program terminated to this specific kernel and which parts are executed for this run. If we do not replicate these functionalities and we are going to terminate to same functionality from different thread from both parallelizations then for ARP profiling, we will be having no clue from which thread we come to this kernel and what we execute for that thread. Therefore, ultimately we will be unable to calculate processing and memories accesses by each thread (processor).

Another reason is **blocktransfer** calls. For **blocktransfer**, we need to mention its source and destination nodes so proper DMA will be assigned the job. Without replication, these **blocktransfer** calls will be unable to work with existing implementation.

As we know, all functional definitions are combined to one header file, **AllInOne.h** (on page 26). All these changes to threads naming convention, replicated functionalities and changes for their naming convention should be echoed with newer functional definitions.

4.4.3.4 Adjusting ARP instrumentation

As explained before, Atomium Record & Playback instrumentation is necessary for our profiling intentions. To keep track of all execution paths, ARP records all these paths followed by running application. ARP points to its path nodes or kernels by **arp_enter()** and **arp_leave()** functionalities. These functionalities keep track of originating thread id, se we need to adjust for newly added threads and replicated functionalities for these threads. The ARP calls inside these functionalities should mark correct originating thread id to avoid bogus profiling results.

4.4.3.5 Updating platform information

The MPA tool consumes sequential code, parallelization directives and platform specs as input to generate parallelization code (on page 15). The platform specs are later on used by High Level Simulator to simulate parallelization for the specific platform details (on page 17). To simulate new embedded component on High Level Simulator, we need to update platform specs with accordance to newer details.

Platform specs are described in a file named **platform.plaf**. The description will include number of processors, memories, DMAs. Each memory will be described in terms of id, read page hit, read page miss, write page hit, write page miss, page size, word depth, and number of ports. Similarly, a DMA specification will be described with its id, maximum pending, number of channels, DMA overhead, processor overhead, cycles per word, line setup. The **platform.plaf** will reflect all changes and alterations made while configuring the newer component. DMAs were not configured at that stage, so now need to create DMAs for each virtual connection established earlier.

4.4.3.6 Block Transfers and FIFO communication

Block transfer are used to communicate data for L1→L2 and L2→L1 transactions. We dumped threads and their relative functionalities codes here, but need to settle down all block transfers, because processor id will be different. Secondly, block transfer id should be unique and may now have a clash with other parallelization block transfer. Therefore, for block transfer we need to adjust block transfer id, source and destination ids and virtual connection id the block going to use.

For FIFO communication, FIFOs are configured fully while configuring component, so at this stage while communicating through a FIFO one just need to mention FIFO id. Therefore, here we need to adjust the FIFO ids so communication can be done through proper FIFO.

4.4.3.7 Execution and Profiling

After all these major adjustments and some other minor adjustments, we are ready with our newer component having two parallelizations coupled. For inter-frame switching, we will select parallelization

before start of each frame processing for input sequence. Now we have choice to encode complete or part of sequence with any parallelization. By encoding complete sequence with only one parallelization, we will receive same results, as achieved earlier with that parallelization. Each frame takes around 5 to 8 million cycles for encoding and cycles required for thread spawning, thread activation and argument passing will require few thousand cycles. Therefore, for total sequence encoding, cycles required for thread spawning and activation are negligible and can be ignored. For a 10 frames sequence, we either switch for each frame or switch just once after 5 frames, both parallelization will encode half of sequence frames and profiling results will be same. Even previously generated parallelization, spawn and activate thread for each sequence frame, so switching from one parallelization to other at start of each frame, does not affect much total simulation cycles utilization. The profiling results shows that, we are not just able to switch between previous Pareto points formed by both parallelization, but may also have as many Pareto point to settle down in between these points as many frames the input sequence consists of.

Chapter 5 Results

The ANL profiling provides detailed insight to MPEG-4 encoder application. These results are helpful to understand application behavior. From overall cycle distribution presented in Figure 19, it can be concluded that identified potential functionalities for parallelization in previous work are quite promising. While proposing parallelization possibility, keeping **me** and **mc** functionality on two separate thread was not supportive because of large memory accesses. The two selected parallelizations are embedded to one component to access performance-cost correlation of both parallelizations at run time. The successful experiment shows that we are not just able to access pervious performance-cost instant on Pareto curve, but may also mix up both instants and generate a lot of instances to offer in-between previous ones.

5.1 Interpreting ANL profiling results

The profiling results generated by ANL are huge, giving detailed array accesses of around 150 functionalities. Functionalities consuming most of total simulation cycles are filtered and their array accesses are analyzed. The detailed cycle distribution in form of code flow is shown in Figure 19.

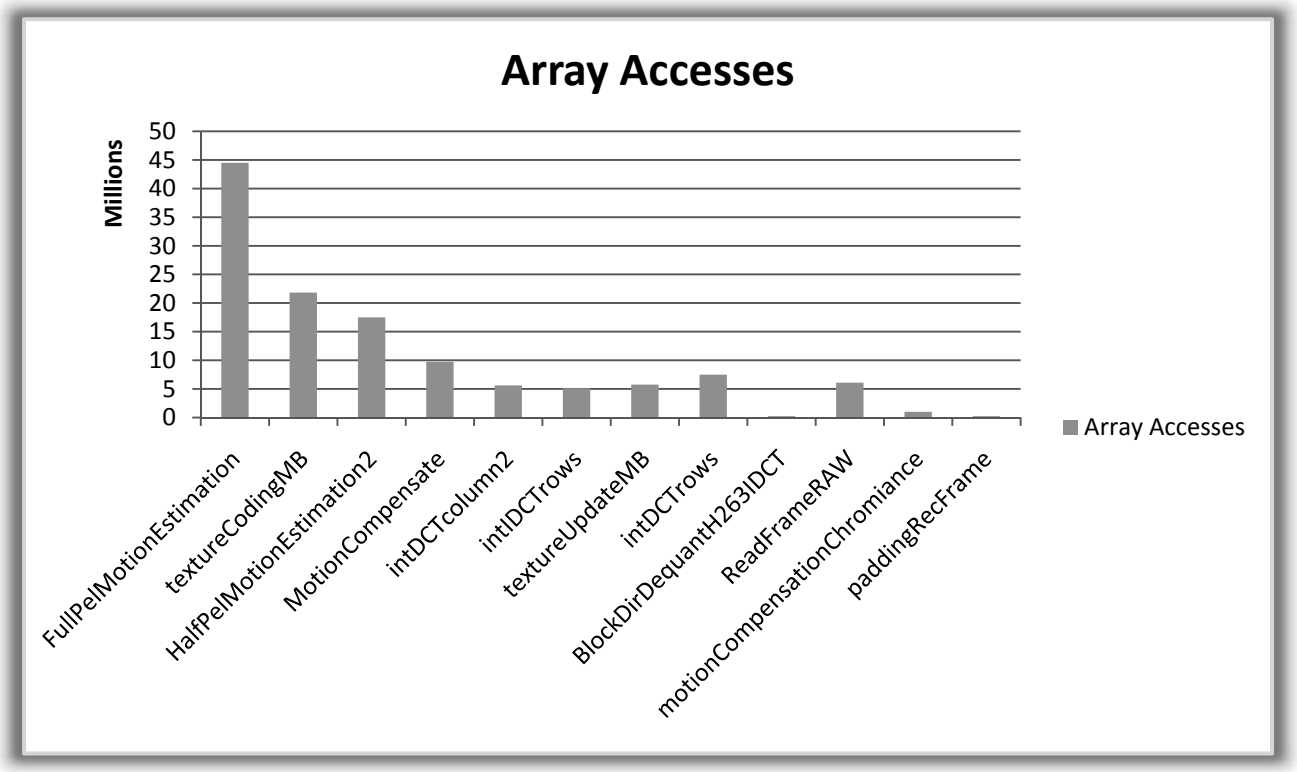

Figure 26 Total array accesses for selected functionalities (read + write).

The Figure 26 shows local array accesses done by selected functionalities. Previously decided parallelizations which are based on simulation cycle distribution, are in agreement with memory access distribution too.

One important point to notice here is how much memory accesses we are covering within these selected functions, which are in numbers 8% of total functionalities. The Table 1 shows that above selected functionalities cover 80% of total memory accesses. The ANL profiling gives us a deep insight into application behavior, simulation cycle distribution and memory accesses distribution. This information is very crucial to for deciding parallelization opportunities available for target application and can be helpful for adding more possibilities to previous parallelization proposals.

Total Array Accesses	146,506,002		
Accesses Covered in Selected Functions	117,805,054		
Accesses not covered	28,700,948	% Accesses Covered	80.40971182
Total Array Writes	50,296,305		
Writes Covered in Selected Functions	40,582,892		
Writes not covered	9,713,413	% Writes Covered	80.68762109
Total Array Reads	96,209,697		
Reads Covered in Selected Functions	77,222,162		
Reads not covered	18,987,535	% Reads Covered	80.26442698

Table 1 Total memory access coverage by selected functionalities

5.2 Profiling Parallelization switcher component

The parallelization switcher component needs to be explored for all possible combination of parallelization selection. For a 10-frame sequence, we can profile with 11 different combinations and we always want to compare our performance gain with sequential execution of application to have a better observation about speedup factor.

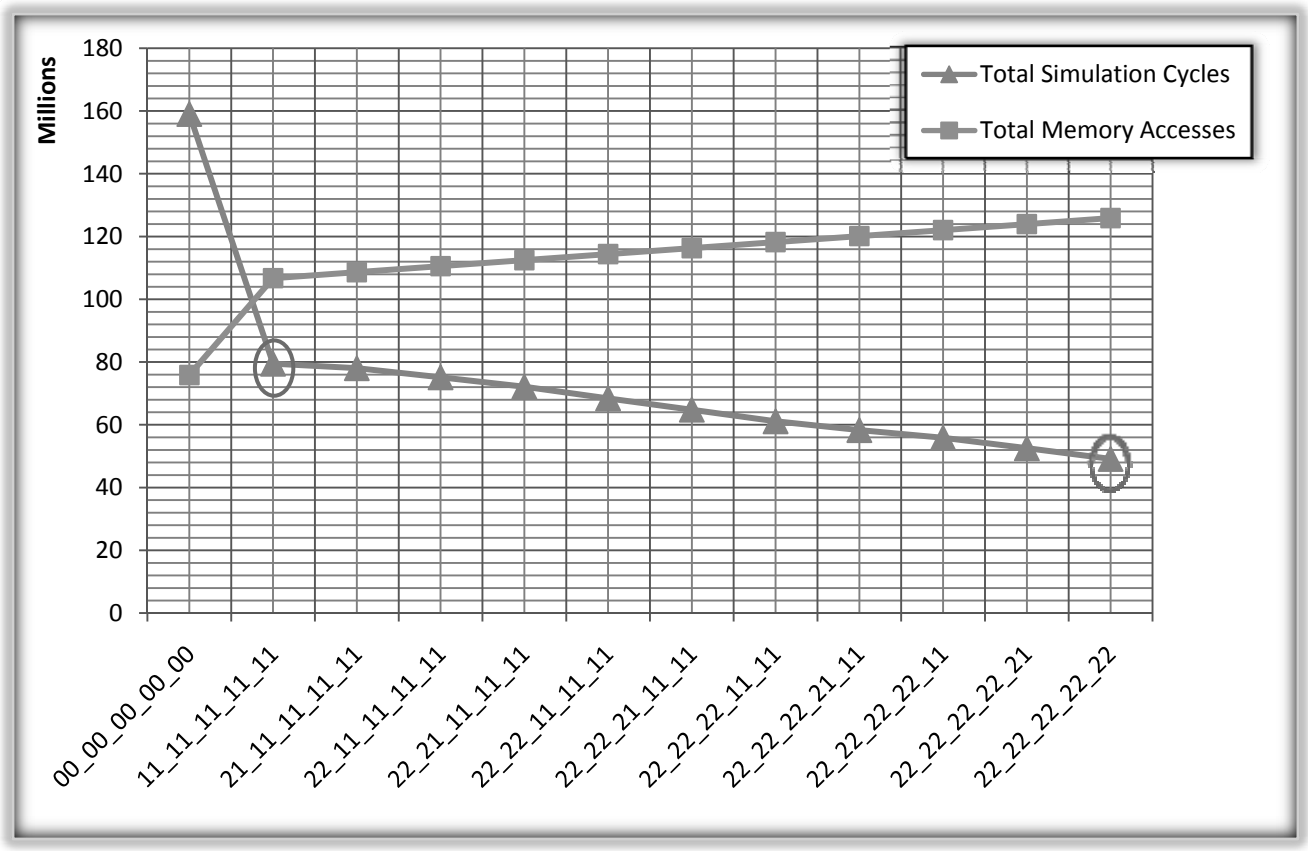

Figure 27 Profiling Results for Parallelization Switching Component (Crew 4CIF sequence)

In Figure 27 “0” represents no parallelization, means sequential or one thread execution. The “1” represent 3-threaded parallelization **memc_tctu_vlcbs**, while “2” represents 7-threaded parallelization **3memc_2tctu_vlc_bs**. The numbers under each entry represent order of frame execution by parallelization option e.g. “22_22_11_11_11” shows that first 4 frames are encoded by 7-threaded parallelization and rest 6 are encoded by 3-threaded parallelization.

The starting Pareto points are marked with red circles. The resultant curve offers many more points in between previously defined points, as our resultant encoding process can be a combination of both parallelizations. For experimental point of view, we took 10-frame sequence but in practical, the number of frames can be in hundreds. The number of newer point in between previous points is n-1, where n represents number of frames in the sequence. Therefore, for a long sequence we will be able to explore curve more smoothly. The above experiment gives us run-time control over platform resources. One can adjust the memory bandwidth utilization by application at run-time and subsequently set the desired performance. In a nutshell, we can improve application performance to 39% at run-time by utilizing 15% extra memory bandwidth.

Here, a point needs to emphasis that we do not need to explore for possible permutations but just possible combinations are enough.

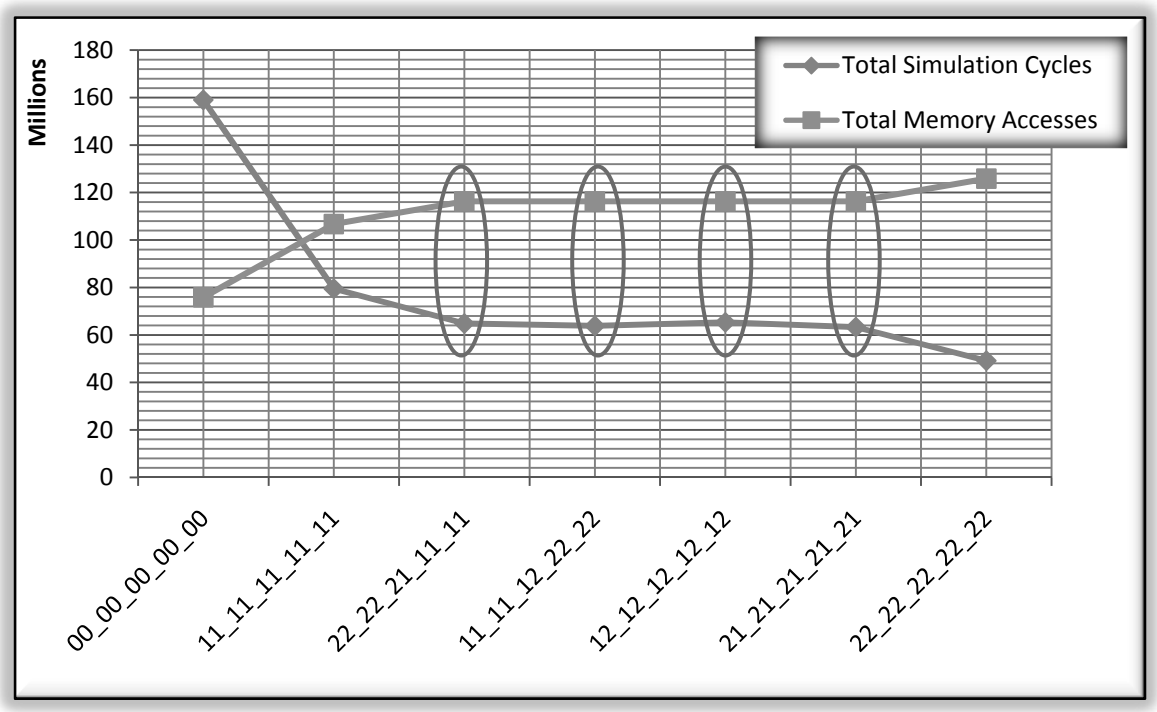

Figure 28 Switching parallelizations cost negligible

We have two strong reasons to support this argument. First, all frames are of almost equal data size, so does not make much difference which parallelization is encoding which frame but how many frames are being encoder by which parallelization. Second, as cost for switching from one parallelization to other is almost negligible in comparison to cost consumed for application execution. Therefore, if we are encoding half of sequence with one parallelization and half with other, it does not matter either we switch after each frame or switch just once in the middle, experiment shown in

Figure 28.

Same experiment is repeated with another sequence to make results non-dependent to input sequence and stress on obtained results. The profiling results for harbor sequence are very much in line with previously generated crew sequence results. For this sequence, 7-threaded parallelization gives much better performance than 3-threaded parallelization and effect can be seen by extra steepness in simulation cycle curve.

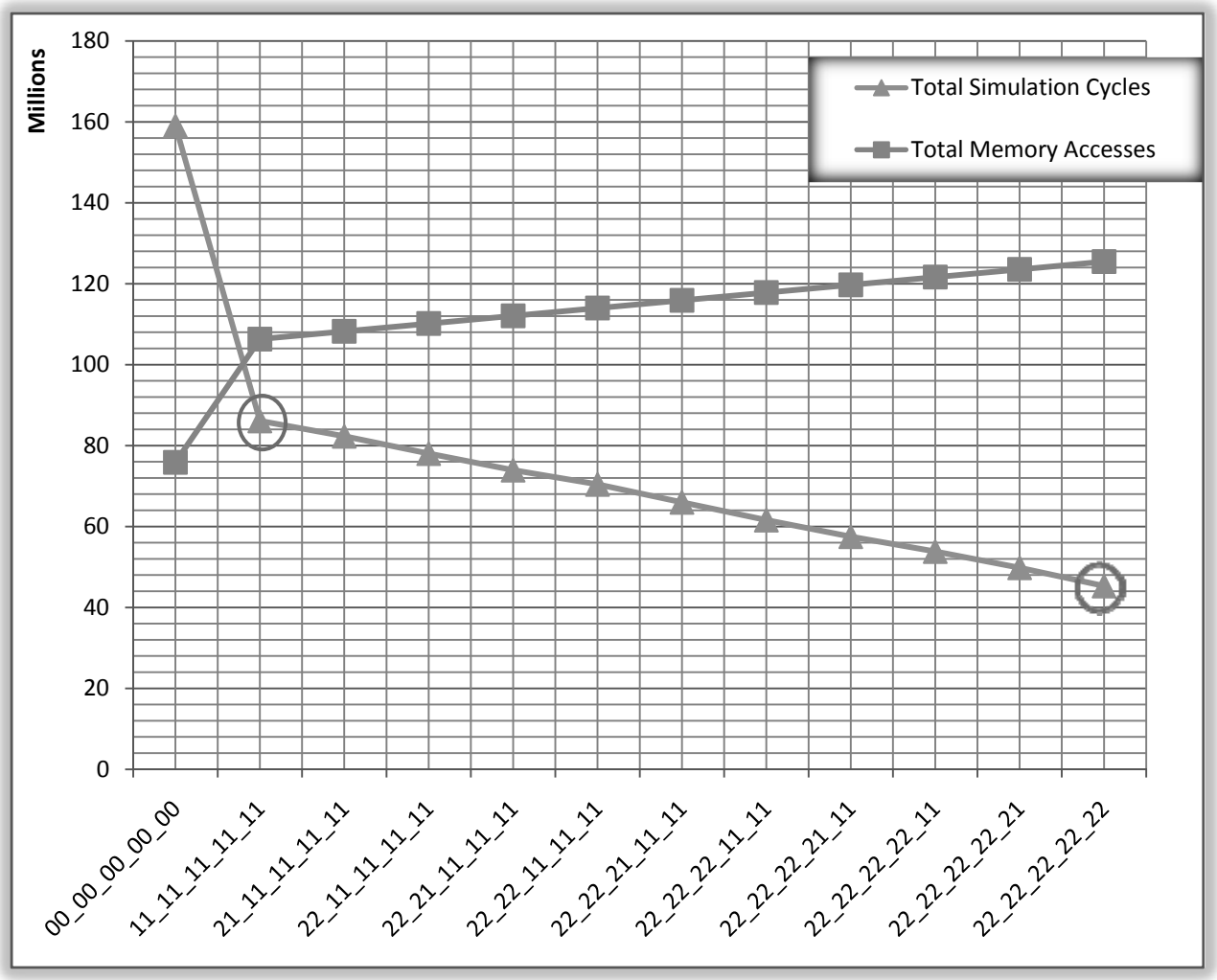

Figure 29 Profiling Results for Parallelization Switching Component (Harbor 4CIF sequence)

Previously known operating points are marked with red circles. Again, sequence was 10-frame sequence and we can find n-1 extra Pareto points to operate in between previously defined operating points. The 3-threaded parallelization consumes 86 million cycles while 7-threaded parallelization consumes 45 million cycles for encoding complete 10-frame sequence. Concisely, we have control to improve our application's performance to 48% at run-time by utilizing 16% extra memory bandwidth.

5.3 The Final Pareto Curve

From the above figures, we may extract the Pareto curve in proper form showing performance and cost relationship. The Pareto curve shows trade-off between performance and cost.

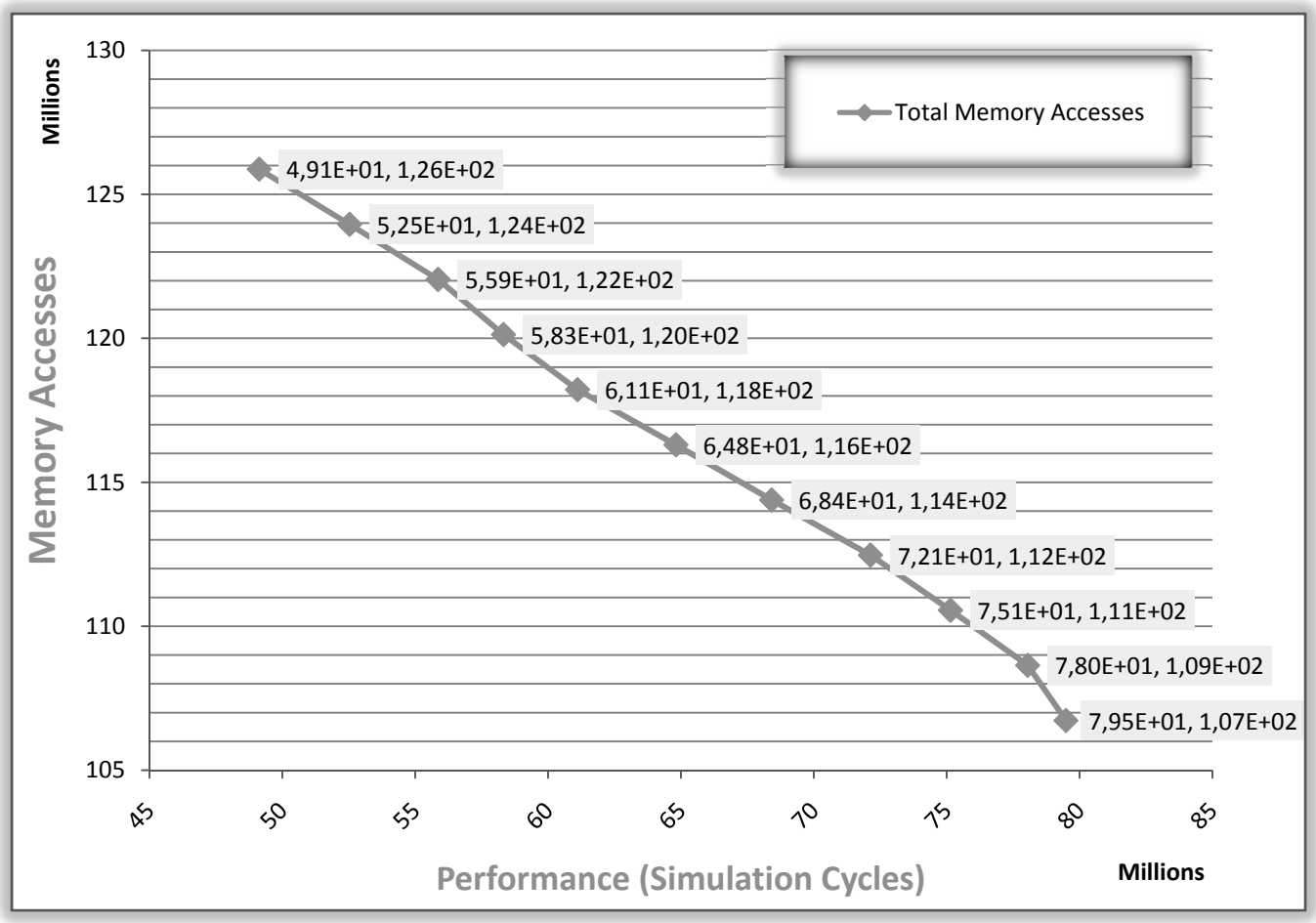

Figure 30 Performance-Cost Pareto Curve (Crew 4CIF 10-frame Sequence)

The Figure 30 shows performance- cost relationship for crew 4 CIF sequence. One can explore this trade-off curve at run-time. The performance of application can be improved or trimmed from 40 to 50% and in this way memory bandwidth utilization can be consumed or saved to 15-16%.

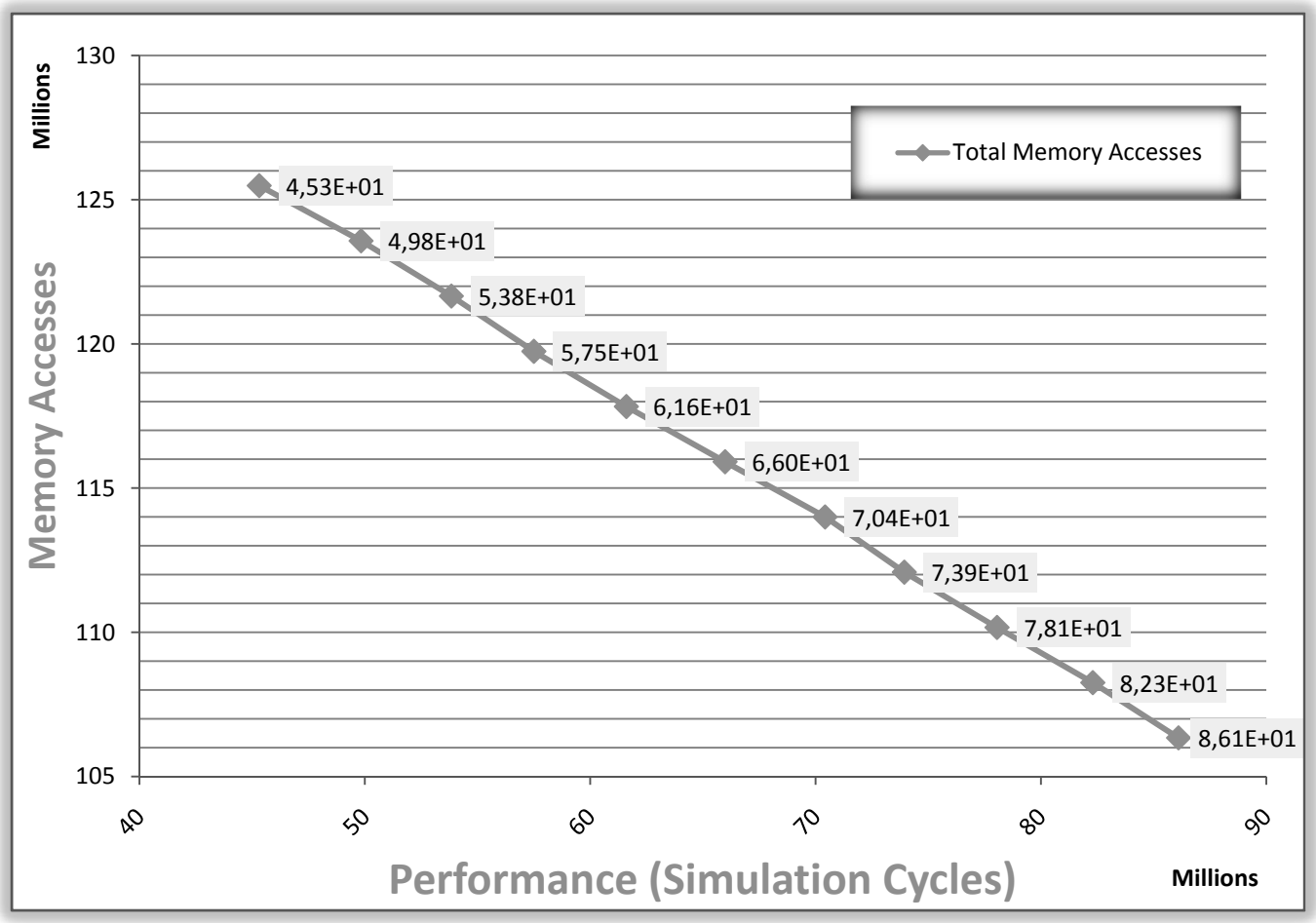

Figure 31 Performance-Cost Pareto Curve (Harbor 4CIF 10-frame Sequence)

Chapter 6 Conclusions and Future Work

In MPSoC environment running multiple applications, resource management (Distribution, allocation and arbitration) plays a key role in system flexibility. The platform should be able to recognize the change in user priorities for running applications and accordingly reflect change in application run-time quality. The run-time parallelization switching enables the platform to echo modification in user requirements/constraints at run-time.

6.1 Conclusions

The results presented shows that parallelization switching enables us to tune application performance at run-time. Adjusting application performance at run-time also enables to have control over bandwidth utilization by application.

Test application and selected parallelization shows that switching between two parallelization of same application does not consume extra cost and a marginal adjustment in form of performance and bandwidth utilization is possible.

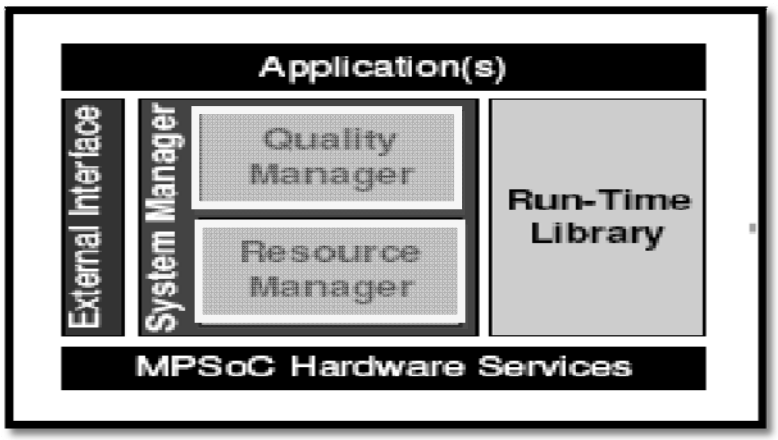

Figure 32 The run-time manager[10]

The figure above shows that thesis work targeted system manager part of run-time manager and both quality & resource manager are capable to arbiter application's quality at run-time.

6.2 Future Work

As thesis work was experiment for run-time control over system resources. Further dig into system is required to achieve the ultimate benefits of thesis work. The future goals can be:

1. As with current research two parallelization for same application were selected to experiment parallelization switching, further goal can be to switch between more than two Pareto points. This will give more flexibility to control application's performance and application quality can be tuned more smoothly.
2. For current work, parallelization were embedded into single executable through manual work. To avoid this tedious work for each application, which isn't a favorable option for practical implementations, parallelization embedding can be made fully automatic or semi-automatic. So at design time, developer will select preferred Pareto points and resultant executable will contain all those parallelization with automatic generation.
3. As the final goal is to run multiple applications on MPSoC and run-time manager is resource arbiter among running applications. So parallelization switching can be experimented with multiple applications, each application have multiple Pareto points to switch. In this way, at run-time user may provide priority to any application and run-time manager will mirror this priority by tuning high performance Pareto point for selected application and similarly other running application will be switched to lower performance Pareto points.

REFERENCES

1. http://www.imec.be/design/mpsoc/mpsoc_dtm.shtml, *MPSoC - Design Time Application Mapping.*
2. http://www.imec.be/design/mpsoc/, *MPSoC Activity.*
3. http://www.imec.be/design/mpsoc/mpsoc_arch.shtml, *MPSoC - Platform Architecture Exploration.*
4. http://www.imec.be/design/mpsoc/mpsoc_rtm.shtml, *MPSoC - Run-time Platform Management.*
5. *IMEC - Scientific Report Year 2006.* 2006.
6. Bingfeng Mei, S.V., Diederik Verkest, Hugo De Man, Rudy Lauwereins, *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix.* 2003.
7. Mladen Berekovic, A.K., Bingfeng Mei, *Mapping MPEG Video Decoders on the ADRES Reconfigurable Array Processor for Next Generation Multi-Mode Mobile Terminals.*
8. Bingfeng Mei, S.V., Diederik Verkest, Hugo De Man, Rudy Lauwereins, *Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matriz Architecture: A Case Study.* 2004.
9. Rogier Baert, E.d.G., Erik Brockmeyer, *An Automatic Scratch Pad Memory Management Tool and MPEG-4 Encoder Case Study.* 2008.
10. Vincent Nollet, D.V., Henk Corporaal, *A Quick Safari Through the MPSoC Run-Time Management Jungle.* 2007.
11. C. Couvreur, V.N., T. Marescaux, E. Brockmeyer, F. Catthoor and H. Corporaal, *Design-time application mapping and platform exploration for MPSoC customized run-time management.* IEE Computers & Digital Techniques, March 2007.
12. S. Khan, K.L.a.E.M., *The utility model for adaptive multimedia systems.* 1997.
13. C. Couvreur, V.N., F. Catthoor and H. Corporaal, *Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MPSoC Run-Time Management.* 2006.
14. Nollet, V., *IMEC MPSoC Project White Paper*

Rationale, Programming Model and Tools.

15. Arnout Vandecappelle, E.D.G., Sven Wuytack, *clean C for MPSoC.* 2007.
16. F. Catthoor, e.a., *Custom Memory Management Methodology.* 1998.
17. K. Denolf, C.B., G. Lafruit and J. Bormans, *Intial Memory complexity analysis of the AVC codec.* 2002.
18. The_Atomium_Club, *Atomium/Analysis 3.3.2 User's Manual.* 2008.
19. The_MPSoC_Team, *MPSoC Parallelizing Assistant User's Manual.* 2007.
20. The_MPSoC_Team, *HIGH-LEVEL PLATFORM SIMULATOR Users and Reference Manual.* 2008.
21. Martin Palkovic, R.B., *Mapping of data intensive applications on multi-processor platforms.*
22. Erick Brockmeyer, R.B., *Mapping of MPEG-4 encoder using MH and MPA showing the MPMH potential.* 2008.
23. Eddy De Greef, e.a., *Mapping and platform exploration flow using Memory Hierarchy and Multiprocessor Parallelization Assistant - tool synergy enhancement.* 2008.