Application Analysis for Parallel Processing

Muhammad Rashid
Thomson Silicon Components,Rennes, France.

Université de Bretagne Occidentale,
CNRS, UMR 3192, Lab-STICC, Brest, France.
muhamad.rashid@thomson.net

Damien Picard and Bernard Pottier
Université de Bretagne Occidentale,
CNRS, UMR 3192, Lab-STICC, Brest, France.
{damien.picard, bernard.pottier}@univ-brest.fr

Abstract

Effective mapping of multimedia applications on massively parallel embedded systems is a challenging demand in the domain of compiler design. The software implementations of emerging multimedia applications are often huge and it is virtually impossible to analyze these applications for parallel processing without generic automated tools.

This paper proposes a two step methodology for specifying multimedia applications (video applications in this paper) in form of parallel process networks. In the first step of the proposed methodology, an input program written in a high-level language is translated into a trace tree representation by dynamic analysis. Operations are performed on the trace tree representation of the application to get analysis results. In the second step, results obtained from application analysis are exploited to re-formulate the application in form of parallel process networks expressed in Avel framework. A well known multimedia application MPEG-2 decoder serves as a case study. Experimental results show the applicability of the proposed methodology.

1. Introduction

The performance gains in multi-core architectures depend on effective application parallelization across cores [1]. For uni-processors, languages like C, C++ were portable, high performance, malleable and maintainable. But for the era of multi-core processors, these languages are not well suited because they assume a single instruction stream and a monolithic memory [2]. The primary goal of these high level source specifications is to provide a reference for verification of a particular functionality. An efficient implementation of these high level specifications on multi-core platforms raises two key challenges: The first challenge is that parallel tasks must be identified and extracted from the sequential reference specification. The second challenge is that there must be an excellent match between the extracted tasks and the architecture resources. Any significant mismatch results in performance loss and a decrease of resource utilization.

In this paper we present a two step methodology for capturing multimedia applications in form of parallel process networks. The first step is application analysis that characterizes source specification of the application at a higher abstraction level without any architectural directives. The input high level specification (written in Smalltalk) is transformed into internal trace tree representation by dynamic analysis of the source specification and analysis operations are performed on trace tree representation. The output of the first step is in form of analysis results. In the second step, analysis results from the first step are used to describe source specification in *Avel* framework. The objective is to describe source specification of the application in form of parallel process networks (stream graph representation) to perform parallel and distributed computations. *Avel* is an architecture independent stream language that exposes inherent parallelism and communication topology of the application enabling the compiler to perform many stream aware optimizations. *Avel* Processes are abstract programmable units implemented with behavioral code of Smalltalk.

This paper is organized as follows: Section 2 describes state of the art in the domain of high level application analysis and stream programming languages. It also summarizes the innovative points of the proposed methodology. Section 3 describes a high level generic analysis framework to extract important characteristics (analysis results) of the application. It also describes experimental results for MPEG2 decoder [3] algorithm (2D Inverse Discrete Cosine Transform, Huffman decoding) to illustrate the analysis methodology. Section 4 summarizes salient features of the *Avel* framework. Section 5 describes the implementation of MPEG-2 decoder with *Avel* framework. We conclude the paper in section 6.

2 Review of related work and contributions

We have divided our related work into two different parts according to two different phases of the proposed methodology. The first phase is to analyze the high level source specification to obtain analysis results and the second phase is to exploit analysis results (obtained in the first phase) to describe the application in form of parallel process networks.

2.1 Application analysis

In [4], some state-of-the-art high-level application analysis approaches for multimedia system design have been comprehensively reviewed including the academic and commercial frameworks. We have further subdivided our related work in this category into application analysis techniques and spatial parallelism.

2.1.1 Application analysis techniques

It may be static analysis or dynamic analysis. Static analysis techniques yields bounds on run-time best and worst cases [5]. The main drawback of these techniques is that the processing complexity of multimedia algorithms heavily depends on the input data statistics while static analysis can only detect upper and lower bounds [6]. In dynamic analysis , alternative solutions are available for tracing a program behavior . It includes source code modification , byte code modification, instrumenting the virtual machine and method wrappers [7][8].

2.1.2 Spatial parallelism

In [11], a profile based technique is presented to extract parallelization from a sequential application. It transforms source specification into a graph based representation to identify parallelizable code. As it measures memory dependencies between different functions of the application so granularity of extracted parallelism is larger and may not be well suited to extract fine grain parallelism. SPRINT [12] tool automatically generates an executable concurrent model in SystemC starting from sequential C code and user defined directives. Again this tool only extracts functional parallelism and leaves the extraction of data parallelism. Commit research group from MIT presents a framework [2] that exposes task, data and pipeline parallelism present in an application written in StreamIt [13] (streaming programming language). It further explains that it is not necessary that all parallelism has equal benefits so it is critical to leverage right combination of task, data and pipeline parallelism.

2.2 Streaming languages

The idea of language dedicated to stream processing is not new and has already been discussed in existing literature[14]. The languages of recent interests are Cg[15], Brook[16], Caravela[17], StreamIt[13], StreamC[19] & Spidle[18]. Existing stream languages can be divided into two categories.

The first type of languages are geared towards the features of specific hardware platform such as Cg[15], Brook[16], & Caravela[17]. All of these languages are dedicated to programming GPUs. The Cg[15] language is a C-like language that extends and restricts C in certain areas to support the stream processing model. Brook[16] abstracts the stream hardware as a coprocessor to the host system. Kernel functions in Brook[16] are similar to Cg[15] shaders. These two languages do not support the distributed programming. Caravela[17] applies stream computing to the distributed programming model.

The second type of languages are limited to introducing a language for gluing components of stream library such as StreamIt [13], Spidle[18], and StreamC[19]. StreamIt [13] and Spidle[18] are stream languages with similar objectives. However, the former is more general purpose while the latter is more domain specific. StreamIt[13] is a Java extension that provides some constructs to assemble stream components. However, a common problem with StreamIt[13] is that each channel can only carry one type of data. Spidle[18], on the other hand is domain specific language for specifying streaming applications. It provides high level and declarative constructs. [19] with a syntax similar to C and is used to define high level control and data flow in stream programs. The stream C compiler analyzes stream program and applies knowledge of high level program structure to stream hardware.

2.3 Contributions

The main innovations of our work versus state of the art tool technology can be summarized as follows.

2.3.1 Generic application analysis

In most of the cases , analysis results are summarized/restricted to only some special design metrics [9][10][11]. The proposed analysis framework is generic and can be extended to fulfill multiple requirements of design space exploration by simply defining new operations on trace tree representation of source specification.

2.3.2 Parallel and distributed computations

The *Avel* framework is proposed for both embedded parallel processing units as well as distributed computations. *Avel* is

not biased towards the features of a specific hardware platform. *Avel* framework has been developed in Smalltalk environment thus we can benefit from reflective properties of the language .

3 Application analysis framework

Figure 1 shows the proposed framework which is divided into two parts. The first part is related to transformation of source specification of application (written in Smalltalk) into a trace tree representation. The second part is related to analysis of the trace tree to get analysis results.

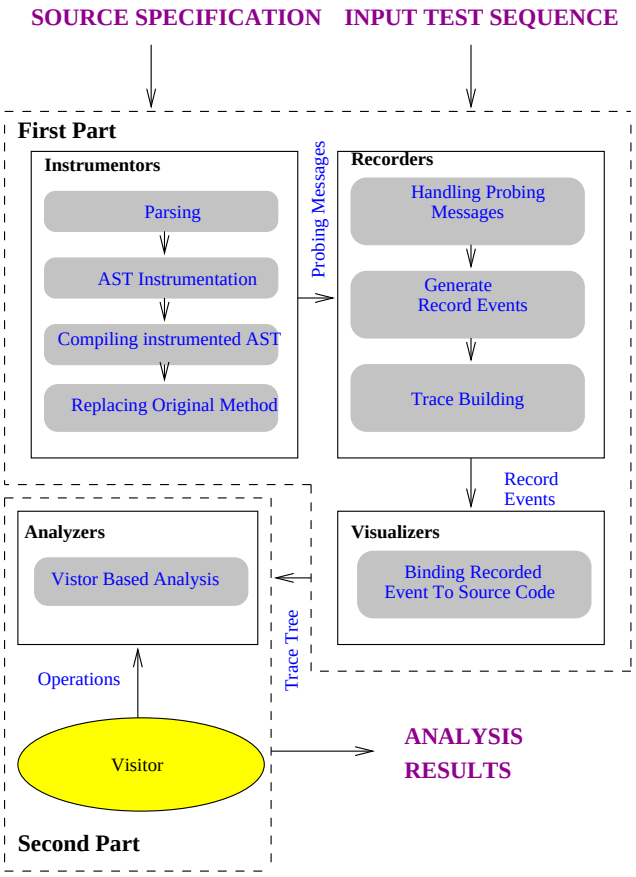

Figure 1. Application analysis framework

3.1 Source specification transformation

The first part is to transform the application into trace tree representation by dynamic analysis of application. Trace tree contains information about execution of the application at run time representing implementation independent application characteristics.

3.1.1 Steps of transformation flow

First part of figure 1 summarizes the required steps of source specification transformation into the trace tree. These steps are [20]:

- *Instrumentors* generate *probing messages* for *recorders* in four sub-steps. The first sub-step is to parse the source specification and generate an abstract syntax tree (AST). The second sub-step is to instrument the AST to generate a new AST with additional nodes. The output of this sub-step is an instrumented AST. The third sub-step is to compile the instrumented AST. The output of this sub-step is the compiled source code. The fourth and last sub-step is to replace the original source code with the compiled source code.
- The output of *Instrumentors* is in form of *probing messages*. The *Recorder* answers all *probing messages* and creates *recorded events*. These *recorded events* are *RecordBlock*, *RecordItem*, *RecordMethod*, *RecordVariable* and *RecordAssignment*.
- *Visualizers* are responsible to bind each *recorded event* (*RecordBlock*, *RecordItem*, *RecordMethod*, *RecordVariable* and *RecordAssignment*) to original Smalltalk source specification in the form of trace tree.

3.1.2 Trace tree representation

The output of the first part of the framework is a trace tree which represents sequence of recorded events, in a tree-like form. We perform analysis operations on the trace tree for different types of analysis. There are several analysis operations that can be performed on the trace tree representation of source specification. For example:

- Checking the value assigned to each variable in each step of the program execution.
- In the context of code rewriting, operations are performed for type-checking.
- To find application orientation in form of processing, memory and control oriented operations.
- To find spatial parallelism at each hierarchical level of the trace tree for every function in the application.

3.2 Spatial parallelism

Trace tree of a particular function or the complete application shows inherited spatial parallelism among operations of the function or application. In other words, inherited spatial parallelism present in the application can be

exploited. We represent the amount of average inherited spatial parallelism for every function in the source specification by *P* such that greater the value of *P*, greater will be the amount of inherited spatial parallelism and vice versa. *P* enables the classification of application functions according to their capability to exploit the inherited spatial parallelism. The value of *P* at any hierarchical level of a trace tree is computed by dividing the total number of operations (RecordItems in a trace tree) by its *Critical Path* . The *Critical Path* at any hierarchical level of a trace tree is the number of longest sequential chain of operations (processing, control, memory). When we compute the value of *P* for a hierarchical level in a trace tree, we assume that the parallel execution of sub hierarchical levels is possible and the value of *P* is given as the ratio between the sum of all operations in the sub hierarchical levels of the node and the longest of all the critical paths.

3.3 Experimental results

In this section analysis results of some parts of MPEG2 decoder application implemented in Smalltalk are presented to illustrate proposed analysis methodology.

3.3.1 Inverse discrete cosine transform

Table 1 shows analysis results for different functions in 2D IDCT. From a structural point of view, 2D IDCT is composed of two identical and sequential 1D-DCT sub-blocks (operating on rows and columns), so the corresponding trace trees have same orientation values for both functions as shown in table 1. The first observation is that the percentage of control operations is zero for all functions, since it is composed of deterministic loops and does not contain any test. Secondly we observe that computation percentage (abbreviated as Comput. in table 1) for 2D IDCT functional blocks are higher so it is computation oriented. The results also show a good percentage of memory operations. We can notice that at the lowest level of granularity (1D-DCT sub-blocks operating on rows and columns), the value of P is 1 indicating no fine grain spatial parallelism. It shows that these sub blocks are sequential in nature and does not contain any inherited parallelism. However, parallelism increases at higher level of granularity (2D IDCT). The value of P at this level is 24.14 indicating that a coarse grain parallelism is available.

3.3.2 Huffman decoding

Table 2 shows analysis results for representative functions of Huffman Decoding. These functions have relatively high percentages of control operations denoting heavily conditioned dataflow. The percentage of computation operations also indicates an important computation frequency. There are less number of memory operations as compared to computations and control operations. It indicates that these functions are control and computation oriented. We have not shown the value of P in table 2 because the value of P remains 1 at all hierarchical levels of trace tree.

Table 1. Orientation results for 2D IDCT

Function	Computation	Memory	Control	P
idctCol	76.36	23.64	0	1
idctRow	76.36	23.64	0	1
2D IDCT	77.11	22.89	0	24.14

Table 2. Orientation results for huffman decoding

Function	Computation	Memory	Control
ChroDCDct	49	2	49
CodedBP	52	5	43
LumaDCDct	60	2	38
MBAddrIncr	50	5	45
MBMode:	58.3	8.4	33.3
MotionDelta:	58.2	3	38.8
QScale:	75	0	25
Huff.Decod	60	7	33

4 Avel framework

The second step in the proposed methodology is to describe the source specification of the application in *Avel* framework. The objective is to describe the application in form of parallel process networks to perform distributed computations. Processes are abstract programmable units implemented with behavioral code of Smalltalk.

Avel framework specifies three kinds of processes which are composed hierarchically. The first type is the *Primitive Process* which is the leaf of process network hierarchy and implements an atomic behavior. The second type is the *Node Process* which is a composition of other processes and behaviors. It allows an hierarchical description of process network. The third type is the *Alias Process* which is declared outside the main process and is reused by another process just by a link to its name. We use *Alias Process* to factorize complex behaviors in the code.

4.1 Syntax of Avel Processes

The syntax to declare the *Primitive Process* or the *Node Process* is given as:

Process Name {*Output Connections*} *[Behavior]*

The graphical representation of the *Primitive Process* and the *Node Process* syntax is shown in figure 2 and figure 3 respectively.

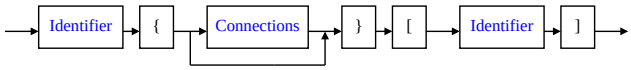

Figure 2. Primitive process syntax

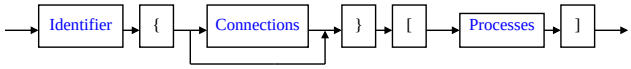

Figure 3. Node process syntax

The syntax to declare the *Alias Process* is given as:

Process Name (process name) [Output Connections]

The graphical representation of the *Alias Process* syntax is shown in figure 4.

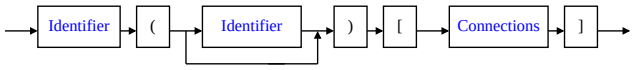

Figure 4. Alias process syntax

The *Process Name* is used as an *identifier* in the process network. To simplify connections between different nodes of the process network, only the *Output Connections* are declared. A connection is specified by names of the nodes and the two ports connected. The graphical representation of the *Output Connections* is shown in figure 5.

The *Behavior* of a process can be atomic or composite. For the *Primitive Process*, the atomic behavior is an *identifier* used to make a link with its corresponding function in the Smalltalk specification. It enables the use of existing library code (written in Smalltalk) in *Avel* processes. For the *Node process*, the composite behavior corresponds to a sub network of processes. The first encapsulated process is connected to the input ports of its hierarchy and the last process is connected to the output ports.

For example, if the output of the process "*ProcessA*" with behavior "*BehaviorA*" is connected to the process

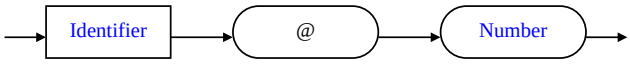

Figure 5. Output connections syntax

"*ProcessB*" at output port "*1*", then it is specified as:

ProcessA {*ProcessB@1*} *[BehaviorA]*

4.2 Avel stream construct example

An *Avel* program (Example) with hierarchical composition of the *Avel* processes is shown below.

```
01.  StrZ{}
02.  [
03.      Prim1{Prim2@1}[Prim1]
04.      Prim2{}[Prim2]
05.  ]

06.  Example{}
07.  [
08.      Split{StrA@1 StrB@1} [splitter]
09.      StrA{StrZ}{Join@1}
10.      StrB{StrC@1}
11.      [
12.          PrimA {PrimB@1}[prima]
13.          PrimB{}[primb]
14.      ]
15.      StrC{StrB}{Join@2 }
16.      Join{}[joiner]
17.  ]
```

The graphical representation of the *Avel* process network in *Example* is shown in figure 6. *Example* (line 6 in above program) is an *Avel* process network that takes an input stream and splits it into two other processes *StrA* & *StrB* (line 8 in above program).

StrZ (line 01 in above program) is a *Node Process* because its behavior contains other *Primitive Processes Prim1* and *Prim2* (line 03 and line 04 respectively in above program). *StrB*(line 10 in above program) is a *Node Process* because its behavior contains other *Primitive Processes PrimA* and *PrimB* (line 12 and line 13 respectively in above program). *StrA*(line 09 in above program) and *StrC* (line 15 in above program) are *Alias Processes* because their behaviors reuse predefined processes *StrZ* and *StrB* respectively. *Split* (line 08 in above program) and *Join* (line 16 in above program) are two *Primitive Processes*: Former is responsible for distributing input stream between its outputs while

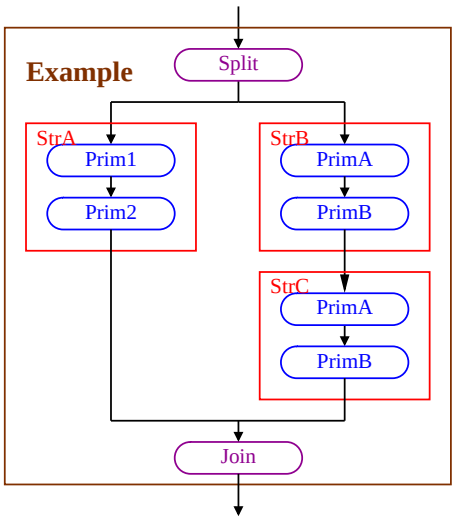

Figure 6. Graphical representation of the AVEL program

the latter is responsible for merging an output stream from its inputs.

4.3 Behavior of Avel process

We implement *Avel* processes with behavioral code of Smalltalk. For example, we present the behavior of *split* process "splitter" as follows:

```
splitter
[
     [true] whileTrue:
     [
          a := in receiveValue .
          out1 sendValue : a .
          out2 sendValue : a .
     ]
] fork
```

The "splitter" behavior splits input stream into two output streams. We can notice that the behavior of *Avel* process is implemented in Smalltalk because the complete *Avel* framework has been developed in Smalltalk environment to benefit from the reflective properties of the language . In other words, the standard Smalltalk code is reused in *Avel* processes. We present another example of behavior of *Avel* process.

```
PrimA[in1] [out1](ST80)
{
01     tmp := in1 receiveValue.
02     res := tmp * 2 .
03     out1 sendValue : res.
}
```

We have directly used inputs and outputs names in the code. An input value is read from the input *in1* (line 01 of the behavioral code) and multiplied by 2 (line 02 of the behavioral code). Then, the result is sent on the output *out1* (line 03 of the behavioral code). *ST80* shows that the behavior is described in standard Smalltalk code.

5 MPEG-2 video decoder in Avel

MPEG-2 video decoding [3] is shown in figure 7.

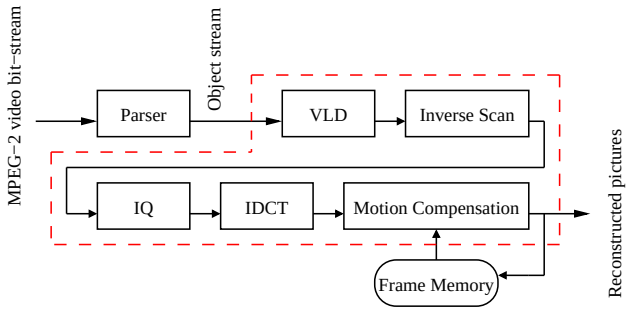

Figure 7. MPEG-2 video decoding process

The basic principle is to remove redundant information from video bit stream at encoder prior to transformation and re-inserting it at decoder. There are two types of redundancies: *Spatial Redundancy* to remove correlation of pixels with their neighboring pixels with in the same frame and *Temporal Redundancy* to remove the correlation of pixels with neighboring pixels across the frames. Each picture is broken down into blocks of size 16x16 luminance samples, called Macroblocks (MBs).

Figure 7 shows that MPEG-2 decoder computations are divided into two parts: The first part is called *Parser* and the objective of the *Parser* is to parse the MPEG-2 video bit stream into the MPEG-2 object stream . In the second part of the MPEG-2 decoder, object stream generated by the *Parser* is used for streaming computations. These streaming computations consist of *Inverse Transformations* and *Motion Compensation*. *Inverse Transformations* are due to the *Spatial Redundancy* reduction and consist of *Variable Length Decoding (VLD)*, *Inverse Scan*, *Inverse Quantization (IQ)* and *Inverse Discrete Cosine Transform (IDCT)*. *Motion Compensation* is due to the *Temporal Redundancy* reduction and performs *Motion Compensation* to recover predictively coded MBs.

5.1 Avel descriptions for parser

The process of parsing the incoming MPEG-2 video bit-stream consists of many layers of nested control flow. It makes the *Parser* unsuitable for streaming computations. It has little in common with stream computations and much in common with context-free grammars. As *Avel* is intended for streaming computations so parsing of MPEG-2 bit stream into object stream is implemented in a higher level language like Smalltalk rather than *Avel*.

5.2 Avel descriptions for picture data

The transformation of MPEG-2 video bit-stream into object stream ensures that all syntactic structures above MBs have been treated. The following *Avel* program shows slice (collection of macro blocks) processing in MPEG-2 decoder . Figure 8 shows the graphical representation of the *ProcessSlice* Avel Program.

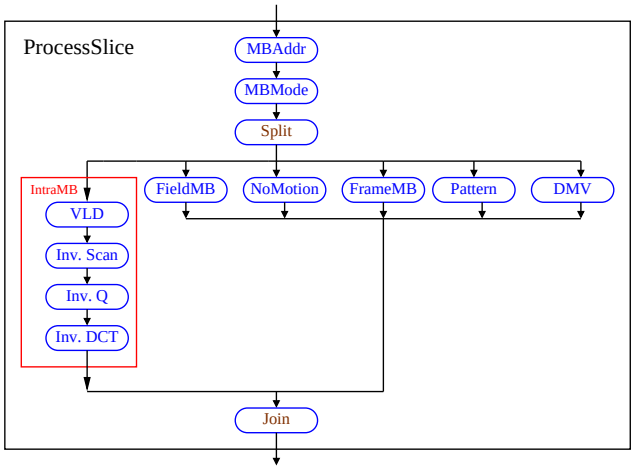

Figure 8. Graphical representation of ProcessSlice Avel program

```
01. ProcessSlice {} [
02.       MBAddr{MBMode@1}[mbaddr]
03.       MBMode {Split@1}[mbmode]
04.       Split {IntraMB@1
05.              FieldMB@1
06.              FrameMB@1
07.              Pattern@1
08.              DMV@1
09.              NoMotion@1}[splitter]
10. IntraMB {join@1}
11. [
12.       VLD{InverseScan@1}[vld]
13.       InverseScan{InverseQuant@1}[is]
14.       InverseQ{IDCT@1}[iq]
15.       IDCT {}[idct]
16. ]
17. FieldMB {} {Join@2}
18. FrameMB {} {Join@3}
19. Pattern {} {Join@4}
20. DMV {} {Join@5}
21. NoMotion{} {Join@6}
22. Join {} [joiner]
23. ]
```

The slice processing (line 01 in above program) starts by calculating MBs address increment (line 02 in above program and shown as *MBAddr* in figure 8). It indicates the difference between current MB address and previous MB address. We have implemented it as a *Primitive Process* with behavior *mbaddr* as shown in figure 8. The behavior of this process contains inherent parallelism. We can implement this process as *Node Process* which contains other *Primitive Processes*. However, we have shown it as a *Primitive Process* in figure 8 for simplicity.

After calculating MB address increment, MB mode (shown as *MBMode* in figure 8 and line 03 in above program) is calculated which indicates the method of coding and contents of the MBs according to tables defined in MPEG-2 standard [3]. Each type of MB is treated differently. Again, we have implemented it as a *Primitive Process* with behavior *mbmode*. However we can implement it as *Node Process* containing other *Primitive Processes*. The output of *MBMode* can be given to any of the six processes as shown in figure 8 (line 04 to line 09 in above program). All of these processes (*IntraMB*, *FieldMB*, *FrameMB*, *Pattern*, *DMV*, *NoMotion*) are *Node Processes* and consists of other *Primitive Processes*. But for simplicity, we have shown only *IntraMB* as *Node Process* (line 10 in above program) and all of other processes are shown as *Primitive Processes* as shown in figure 8 (line 17 to line 21 in above program). *IntraMB* further consists of *Primitive Processes*. These processes are *VLD*, *InverseScan*, *InverseQ* *IDCT* (line 12 to line 15 in above program). Again, we have implemented all of these processes as *Primitive process*. we can implement these processes as *Node Process* which contains other *Primitive Processes* depending upon the amount of spatial parallelism obtained from analysis framework.

6 Conclusions & Future work

This paper has presented a two step methodology to specify streaming applications as parallel process networks. In the first step, high level application in Smalltalk is transformed into a trace tree representation by dynamic analysis.

Analysis operations are performed on the trace tree representation to extract valuable information (analysis results) about application. The second step in the proposed methodology is to model the source specification by using analysis results from the first step in *Avel* framework. We have described computational intensive part of MPEG-2 decoder using *Avel* framework.

The preliminary experiments of the proposed framework encourage us to extend this work in several directions. It includes automatic transformation of high level source specification into *Avel* process networks. The CDFG model can be produced from *Avel* descriptions that will be an input to existing synthesis tools.

Acknowledgments

We acknowledge the contribution of Thierry Goubier for his development in application analysis framework.

References

[1] Will Eatherton, "The push of network processing to the top of the pyramid.", In *Symposium on Architectures for Networking and Communications Systems*, NewJersey,USA, 2005.

[2] Michael I. Gordon, William Thies, and Saman Amarasinghe, "Exploiting coarse grained task, data, and pipeline parallelism in stream programs.", In *Oper. Syst.*, 151-162, 2006.

[3] Information technology - coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbit/s. *ISO/IEC 13818-3:International Organization for Standardization*, 1999.

[4] Matthias Gries. "Methods for evaluating and covering the design space during early design development.", In *VLSI Journal*, 38(2):131-183, 2004.

[5] P. Puschner and Ch. Koza., "Calculating the maximum, execution time of real-time programs.", In *Real-Time Syst.*, 1(2):159-176, 1989.

[6] S. Mallat and F. Falzon., "Analysis of low bit rate image transform coding.", In *IEEE Transactions on Speech, and Signal Processing*, 46(4):1027-1042, 1998.

[7] M.Denker, O.Greevy, and M.Lanza., "Higher abstractions for dynamic analysis.", In *(PCODA06).*, IEEE, 2006.

[8] A. Hamou-Lhadj., "The concept of trace summarization." , In *(PCODA05)*, pages 43-47. IEEE, 2005.

[9] M. Ravasi and M. Mattavelli., "High abstraction level complexity analysis and memory architecture simulations of multimedia algorithms" In *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):673-684, May 2005.

[10] Y.Moullec, Jean-P. Diguet, Nader B. Amor, T. Gourdeaux, and Jean L. Philippe., "Algorithmic-level specication and characterization of embedded multimedia applications with design trotter." In *VLSI Signal Process. Syst.*, 42(2):185-208, 2006.

[11] S.Rul, H.Vandierendonck, and K. D. Bosschere., "Function level parallelism driven by data dependencies.", In *SIGARCH Comput. Archit. News*, 35(1):55-62, 2007.

[12] J.Cockx, K.Denolf, B.Vanhoof, and R.Stahl., "Sprint: A tool to generate concurrent transaction-level models from sequential code.", In *EURASIP Journal on Advances in Signal Processing*, Article ID 75373, 2007

[13] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe., "MPEG-2 decoding in a stream programming language.", In *IPDPS 2006*, April 2006.

[14] Robert Stephens., "A survey of stream processing.", In *Acta Informatica*, volume 34, number 7, pages 491-541, 1997.

[15] R. Mark, R. Glanville, K. Akeley, and J. Kilgard., "Cg: a system for programming graphics hardware in a c-like language.", In *ACM SIGGRAPH 2003 Papers*, pages 896-907, New York, USA, 2003.

[16] Nolan Goodnight, Rui Wang, and Greg Humphreys., "Computation on programmable graphics hardware.", In *IEEE Comput. Graph. Appl.*, 25(5):1215, 2005.

[17] S. Yamagiwa and L. Sousa., "Caravela: A novel stream-based distributed computing environment.", In *IEEE, Computer*, Volume 40, Pages 70-77, 2007.

[18] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu., "Spidle: A dsl approach to specifying streaming applications.", In *proceedings of 2nd international conference on generative programming and component engineering*, pages 1-17, 2003.

[19] Abhishek Das, William J. Dally, and Peter Mattson., "Compiling for stream processing.", In *PACT 06*, pages 3342, New York, USA,2006.

[20] Muhammad Rashid and Thiery Goubier and Bernard Pottier, "A high level generic application analysis methodology for early design space exploration.", In *DASIP '07:*, Grenoble, France, 2007.