A High Level Generic Application Analysis Methodology For Early Design Space Exploration

Muhammad Rashid
Thomson Silicon Components
Rennes, France
&
Université de Bretagne Occidentale
Brest, France
Email: muhamad.rashid@thomson.net

Thierry Goubier
Université de Bretagne Occidentale
Brest, France
Email: thierry.goubier@gmail.com

Bernard Pottier
Université de Bretagne Occidentale
Brest, France
Email: pottier@univ-brest.fr

***Abstract*— The software implementations of sophisticated multimedia applications/algorithms are often huge and it is virtually impossible to analyze these applications without generic automated tools and appropriate methodologies. Architectural implementation choices for these applications based on merely designer experience without objective measures can lead to costly re-design loops. Application analysis at algorithmic level can produce a variety of useful information providing valuable design space exploration indications.**

We present a generic application analysis approach based on instrumentation based profiling for early design space exploration. First we transform the source specification of application implemented in a high level language (Smalltalk in this paper) into an internal trace tree representation by dynamic analysis. The trace tree of the source specification is then characterized/explored at the algorithmic level. The results of characterization provide guidelines to the designer to select target architecture(s) for the application. These guidelines include memory, control and processing orientations as well as the inherited spatial parallelism in the specification. The aim is to improve application architecture matching by bridging the gap between application specification and target architecture.

As a case study, we have taken MPEG-2 decoder implemented in Smalltalk. Experimental results show the applicability of the the proposed methodology for early design space exploration.

I. Introduction

The exponential growth of VLSI technology is yielding more and more powerful multicore architectures enabling massive integration of processing units and fast communication channels on a single chip [1] [2]. A leading example of this industrial trend is the Cell processor from IBM/Toshiba/Sony that has 9 cores [3]. Cisco has described a next-generation network processor containing 192 Tensilica Xtensa cores [4]. The performance gains in these multicore architectures depend on effective application parallelization across the cores. At the same time, the continuous growing effort of developing multimedia standards with higher compression efficiency, better quality of service and more functionalities have resulted in an extremely high level of algorithmic complexity and sophistication [5]. As a result, architectural implementation choices for complex multimedia algorithms based on designer experience without objective measures become extremely difficult or impossible tasks [6]. It may leads to late realization of sub-optimal (or even wrong) solution in the design cycle resulting in costly design iteration. The term algorithmic complexity itself is not well defined [7]. In this paper we consider it in terms of number of arithmetic operations, memory bandwidth, regularity of the algorithm and possibility of parallel processing. Here the term complexity is not used in its strict mathematical definition only considering the size of algorithm minimal descriptions. In a broader sense, we are mainly interested in run-time aspects of algorithm complexity metrics.

This work was carried out to address an industrial problem relevant to the early design space exploration for multimedia applications.

In order to make appropriate choices about architectural implementation of the given application at the beginning of the design cycle, it is desirable to measure and understand the algorithmic characteristics/complexity of an application before starting the design of the target architecture [6]. The objective of application exploration is to obtain approximate measures that can identify classes of candidate architectures for the actual implementation. Consequently, an automated and generic application exploration tool in the early design space exploration is required.

In this paper, we propose a high level generic application analysis framework that characterizes the application at an higher abstraction level without any architectural directives. The proposed framework provides desirable characterization results and can be used as a first step in design space exploration. It bridges the gap between specification of a system and the definition of a target (or a set of target) architecture(s) for that system. This characterization information of the application is obtained very early in the design process and can be used in three different ways.

- When the target architecture is fixed and the focus of analysis is to optimize the application by providing guidelines about the algorithmic choices.
- When the application at higher abstraction level is fixed, and the analysis focus is to provide indications about implementation choices for that application.
- When neither the specification nor the architecture is fixed, the designer can refine both aspects by using

characterization results.

In this paper we focus on the second point by presenting a high-level generic framework for application analysis. The basic idea is to take a standard code as an input (without performing any additional effort) and to analyze it for early design space exploration. The starting point is an executable software specification written in Smalltalk [8] which is a high level language. This input specification is then automatically transformed into internal trace tree representation by dynamic analysis of the specification. By dynamic analysis, we obtain valuable information about the run time behavior of the specification in a form of trace tree that represents implementation-independent specification characteristics and provide information about the inherent characteristics of the application. We then present a generic analysis framework to analyze and explore the trace tree (representation of the source specification). By using the information provided by the analysis results, the designer is guided in his architectural choices since he gets an insight of the application behavior.

The exploration of the design space for embedded systems may have different meanings. The proposed analysis framework is generic in a sense that it may extract multiple characteristics/features of the application depending on a particular analysis requirement by simply defining new analysis operations on the trace tree representation of the application. In this paper, we focus on the features at system level (where the target architecture is not yet defined) including exploitation of spatial parallelism and application orientation of the application in terms of processing, control and memory accesses.

This paper is organized as follows: Section 2 describes state of the art in the domain of application characterization for early design space exploration targeting the multimedia/video field. It also summarizes the innovative points of our framework. Section 3 describes proposed generic analysis framework to extract the important characteristics of the application. Section 4 describes some usage scenarios of analysis results and their significance in design space exploration. Section 5 partially evaluates MPEG2 decoder algorithm [9] (2D IDCT, Huffman decoding) to illustrate the proposed analysis methodology. It provides some examples of the analysis results in terms of computational complexity, data flow and inherited spatial parallelism. Finally, section 6 concludes the paper.

II. Review of Related Work and Contribution

A. Related Work

In [6], some state-of-the-art high-level application analysis approaches for multimedia system design have been comprehensively reviewed including the academic and commercial frameworks. Two main axes are typically recognized: the orientation of the application in terms of processing operations (algorithmic complexity) and amount of inherited spatial parallelism present in the application. So we consider related work in application analysis techniques, application characterization and spatial parallelism.

1) Application Analysis Techniques: It may be static or dynamic. Static analysis techniques yields bounds on run-time best and worst cases [10] [11]. 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 [12]. In dynamic analysis [13], alternative solutions are available for tracing a program behavior [14]. It includes source code modification , byte code modification, instrumenting the virtual machine and method wrappers [15] [16] [17]. Instruction level profiling is also a form of dynamic analysis providing information at a relatively high level of abstraction (at function level). However it does not provide the statistics about the processing operations executed by those functions. The information gathered with profilers strictly depends on the underlying machine and on the compiler optimization. This is against the requirement of high level system design in which complexity evaluation depends only on the algorithm itself [18].

2) Application Characterization: A characterization approach to find the algorithmic complexity is presented in [19] . The source code is instrumented and simulated to collect execution counts that capture the dynamic behavior of the application. Specification characteristics are then computed by statically analyzing the code together with the collected dynamic information. The idea is to explore the design space with the results that are accurate enough to prune out infeasible design alternatives. In [18], authors introduce an integrated tool for the complexity analysis of C reference descriptions. The tool is capable of measuring all C language operators during the execution of algorithms. The tool capabilities also include the simulation of virtual memory architectures extending it to data transfer and storage analysis. Simulator can be configured to provide measurements and performances of the algorithm under study on user configured memory architectures. In [20], application specified in systemC is statically analyzed to get some analysis results as well as simulated to get some dynamic information. By combining the static analysis and simulation results, work load is estimated in form of metrics. Then a HW/SW partitioning tool is driven based on these analysis results. The purpose is to characterize application functions according to three kinds of targets: GPP, DSP and ASIC. In [21], a high level exploration methodology based on C language is presented. Applications are characterized using a hierarchical graph-based representation of the application, resulting in a set of metrics. These metrics characterize the application in terms of memory bandwidth, processing parallelism and relative control/processing/data transfers. In [22], a way of measuring arithmetic complexity of multimedia applications is described. However it only measures the number of arithmetic and control operations and ignores other aspects of complexity like number of memory accesses, possibility of parallel operations and so on.

3) Spatial Parallelism: In [23], a profile based technique is presented to extract parallelization from a sequential application. It transforms the source specification to a graph based representation to identify parallelizable code. As it measures the memory dependencies between different functions of the application so granularity of the extracted parallelism is larger and is not well suited to extract fine grain parallelism. SPRINT [24] tool automatically generates an executable concurrent model in SystemC starting from sequential C code and user defined directives. First, it transforms C code to control flow graph which is further transformed to model the different concurrent tasks. Again this tool only extracts functional parallelism and leaves the extraction of data parallelism. Commit research group from MIT presents a framework [25] that exposes task, data and pipeline parallelism present in an application written in StreamIt [26] (an streaming programming language). It further explains that not all parallelism has equal benefits so it is critical to leverage the right combination of task, data and pipeline parallelism.

B. Contribution

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

1) Source Specification: Almost all the application analysis methodologies cited in this section start with application specifications written in C language. We propose an application analysis framework in which application is specified in Smalltalk. It is a dynamic implicitly-typed language where objects, not variables, carry type information, freeing the programmer from declaring variable types. The combination of a polymorphic, pure object-oriented language which is simple yet powerful with an incremental programming environment and a large robust class library make Smalltalk an attractive choice for building complex systems that must adapt to the changing needs of embedded system design [8].

2) Instrumentation Techniques: All the instrumentation based application analysis approaches for design space exploration instrument the source code but in the proposed framework, parse tree of the source code is instrumented due to its simplicity, control, generality and preservation of original source code semantics. Smalltalk is a reflective programming language, whereby the objects that define the language are themselves built with the language. It allows the programmer to extend the language and environment in a way that is unparalleled in conventional programming environments like C. Consequently, the proposed framework advocates to instrument on a parse tree rather than any other form.

3) Generic Application Analysis: In most of the cases , analysis results are summarized/restricted to only some special design metrics [21] [18] [23]. According to the fact that design space exploration of the embedded systems may have different requirements like extracting inherited spatial parallelism, analyzing application orientation, code optimizations etc, a generic analysis framework is more attractive choice keeping in mind the increasing heterogeneity of applications and architectures [4]. Our analysis framework is generic and can be extended to fulfill multiple requirements of design space exploration by simply defining new operations on the trace tree representation of the source specification.

III. Proposed Analysis Framework

Figure 1 show the proposed framework which is divided into two parts. The first part is related to the transformation of source specification (written in Smalltalk) into an intermediate trace tree representation. The second part is related to analyze or explore the trace tree of the source specification to get desirable analysis results. We describe the two parts of the proposed framework in the following paragraphs without going into implementation details as it is not the main focus of this paper. We refer interested readers to [27] for the implementation details of the proposed framework.

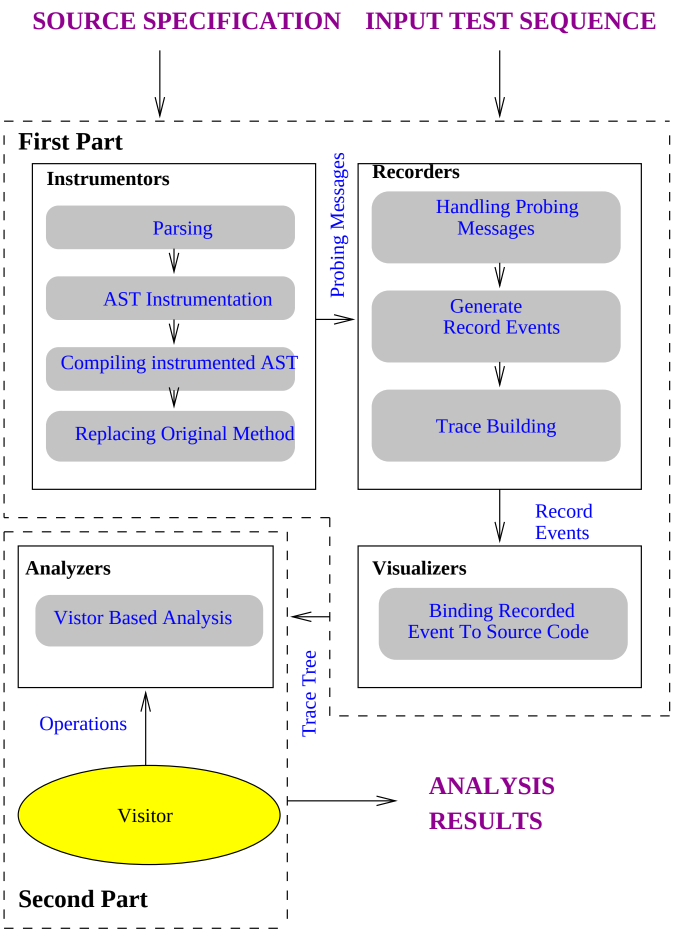


Fig. 1. The Skeleton Of Analysis Framework

A. Source Specification Transformation into Trace Tree

The first step is to write the source specification in Smalltalk enabling the designer to specify and program algorithms at a high level of abstraction (executable specifications) very early in the design cycle. Executable specifications in Smalltalk are high level programs, far simpler than their equivalent implementation on heterogeneous SoCs. In order to transform source specification into a trace tree (intermediate internal representation) that contains information about the execution of an application at run time, dynamic analysis is used as the code instrumentation mechanism that allows insertion of code to monitor and track the runtime behavior of the source specification of the application. It enables to extract valuable information about how the specifications works at runtime [14]. The run time information is recorded in a form of trace tree representing implementation independent specification characteristics. Each function in the application is transformed into trace tree representation. Individual traces for each function are then combined to get the final trace tree of the application that characterizes the complete application [27].

1) Steps Of Transformation Process: First part of figure 1 summarizes the required steps of source specification transformation into trace tree. These sub-steps are:

- *Instrumentors* generate probing messages for recorders in four sub-steps. The first sub-step is to parse source code and generate an abstract syntax tree (parse tree). The second sub-step is to instrument (alter) the parse tree to generate a new parse tree with additional nodes. The third sub-step is to compile the instrumented syntax tree. The output of this sub-step is a compiled method. The fourth and last sub-step is to replace the original source code with the compiled source code. The implementation details of these sub-steps can be seen in [27].
- The output of *Instrumentors* is in the form of probing messages. The *Recorder* answers these messages and create events on reception of probes activation messages. These events are *RecordBlock*, *RecordItem*, *RecordMethod*, *RecordVariable* and *RecordAssignment* as shown in figure 2.
- *Visualizers* are responsible to bind each event to the original source code.

2) Trace Tree: The output of the first part of the framework is a trace tree which represents the sequence of recorded events, in a tree-like form. A typical use of the trace tree is to hierarchically show the structure of function calls. The basic entity of the trace tree is an *Abstract Record* which in turn has its sub entities (children) as shown in figure 2.

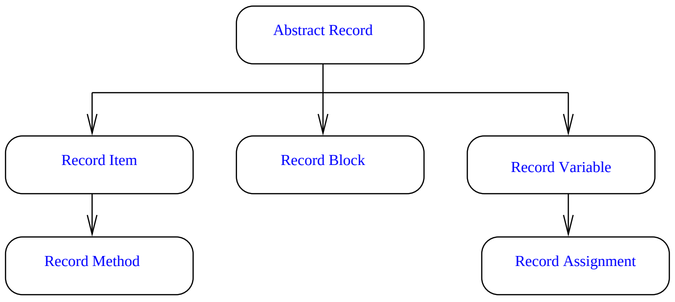


Fig. 2. Record Hierarchy in Trace Tree

- The *AbstractRecord* is the super class of all the nodes in a trace tree. It is an abstract class and does not have any instances. However, all the entities in a trace tree are the subclasses of *AbstractRecord*. It contains the common behavior of all the entities in a trace tree.
- Every operation (computation, memory transfer, control) in a trace tree is recorded as *RecordItem*.
- The top level of every function in a trace tree is recorded as *RecordMethod*.
- Every variable in a trace tree is recorded as *RecordVariable*.
- Every assignment to the variable in a trace tree is recorded as *RecordAssignment*.

The detail description of each entity like *RecordBlock*, *RecordItem*, *RecordMethod* and *RecordVariable* in a trace tree can be found in [27].

B. *Trace Tree Analysis*

Once source specification is transformed into trace tree representation, we perform operations on trace tree for different types of analysis. These analysis operations may be, for example, checking the value assigned to each variable in each step of the program execution. In the context of code rewriting, one may perform operations for type-checking, code optimization, flow analysis and so on. These operations are performed on basic entities like *RecordBlock*, *RecordItem*, *RecordMethod* and *RecordVariable* of the trace tree as shown in figure 2.

We keep basic entities of the trace tree and their subclasses independent of the analysis operations that apply to them by packaging related operations from each class in a separate object named as *visitor* and passing it to elements of the trace tree. There are different analysis purposes and for each analysis purpose, there is a separate visitor.

The proposed analysis framework is generic as it is not restricted to a particular set of analysis operations. It allows the designer to extend the framework by defining new analysis operations (visitors) to fulfill different requirements of design space exploration such that for each analysis operation, there is a corresponding visitor for the trace tree making a visitor hierarchy similar to visitors on a parse tree [28]. The root and all elements of a parse tree are *ProgramNode(s)*, while the basic element of a trace tree is *Abstract Record* as shown in figure 2.

IV. Usage Scenarios of Analysis Results in Design Space Exploration

We have mentioned in the introductory part of this paper that design space exploration for embedded systems may have different requirements due to the growing heterogeneity of applications and architectures [3]. In section III, we have explained that we can perform multiple analysis operations on trace tree representation of the source specification to build a generic analysis framework. Each analysis operation in our proposed framework depends on a particular design requirement of design space exploration. These analysis operations may be for extracting spatial parallelism, analyzing application orientation, type-checking, code optimization and for many other requirements. To illustrate these concepts, we perform analysis operations on trace tree representation of the source specification to extract application characteristics in terms of application orientation [19] [18] [20] and inherited spatial parallelism [21] [23] [25]. In this section, we present a number of the usage scenarios of our analysis results and their significance in the design space exploration.

A. Application Orientation

The orientation of an application gives guidelines about architecture selection. An application may have three types of operations: computations, memory and control. Our analysis results describe the operations in terms of percentages of these three basic types of operations. These types are:

1) Computation Oriented Operations: It includes the arithmetical operations (Addition, Multiplication, Subtraction etc) as well as logical operations (And, Or etc). The higher percentage of this type of operations tells the designer that how much a particular function is computationally intensive. Consequently, designers should more concentrate on computation optimization. There are different types of computation operations. The analysis results show the percentage of each type of computation operations in a function. For example, if most of the operations are multiplications, then target architecture should have dedicated hardware multipliers, hence guiding the designer towards architecture selection.

2) Memory Oriented Operations: The percentage of this type of operations indicates the frequency of memory accesses in a trace tree. The higher percentage of this type of operations tells the designer that a particular function is data access dominated and is most likely to require a high data bandwidth. It indicates that the computations are not performed on previously computed data (reside in local memories) but performed on the input data (data entering to trace tree). Therefore in the case of real time constraints, some efficient mechanism of data movement and high performance memories are required.

3) Control Oriented Operations: The percentage of this type of operations indicates the frequency of control operations in a trace tree. This percentage guides the designer to evaluate the need for complex control structures to implement a function. The functions with high percentage of this types of operations are good candidate for a GPP processor implementation rather than a DSP processor implementation, since the latter is not well suited for control dominated functions. In addition to this, a hardware implementation of these control dominated functions would require large state machines.

B. Spatial Parallelism

Trace tree of a particular function or the complete application shows the existing parallelism among the operations of the function or application. It implies the possibility of mapping different operations/functions to different PEs (processing elements) of the target architecture for concurrent execution. In other words, we can exploit the inherited spatial parallelism present in the application. We represent the amount of average inherited spatial parallelism for every function in the source specification by P [21] such that functions with higher P values are considered as appropriate to architectures with large explicit parallelisms. Functions with lower P value are rather sequential, so the acceleration can only be obtained by exploiting temporal parallelism. P enables the classification of application functions according to their criticality or in other words 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* [21]. The *Critical Path* at any hierarchical level of a trace tree is the number of longest sequential chain of operations (processing, control, memory). It is computed for each hierarchical level.

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. For example, if a node A has three sequential sub nodes B, C and D containing 10, 20 and 30 sequential operations respectively. Now the value of P at each sub node B, C and D is 1 (as they contain only sequential operations and hence no spatial parallelism) but the value of P at node A is 2 (60 divided by 30) assuming that all the sub nodes can be executed in parallel on different execution units. Functions with highest P (spatial parallelism) value can be first considered since they show the most important optimization potential regarding the acceleration.

C. Guidelines for mapping

The mapping process requires application model (in form of different functions) as well as architecture model (in form of processing elements, interconnections etc) to map the application behavior on the architecture model. Our analysis results enable the designer to identify the most complex functions in terms of computations in an application, which may be the best candidates for mapping to the fastest PEs. The designers also prefer to map the functions which communicate heavily (identified by analysis results) to the same PE or to the PEs connected by dedicated busses.

D. Estimation

The performance estimation of different functions of the application on multiple processing elements (PEs) of the architecture is another important issue in the design space exploration. For example, assuming a function F1 is mapped to processing element PE1. If F1 contains X integer-type multiplication operations (revealed by analysis results), and executing such an operation on PE1 requires Y clock cycles (Known to designer from architecture model), then the execution time of function F1 on processing element PE1 will be $X * Y = XY$ clock cycles.

V. Evaluation Results

A. MPEG-2 Decoder

In this section we give analysis results of some parts of MPEG2 decoder application [9] implemented in Smalltalk [29] to illustrate our tracing based analysis methodology. MPEG-2 is a well known encoding and decoding standard for digital video. The basic principle is to remove redundant information prior to transformation and re-inserting it at the 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. The MPEG-2 decoder can be summarized in the following three points [9] :

- *Parser* is responsible for parsing the MPEG-2 bit stream and performing Huffman and Variable run-length decoding (VLD). The input to the parser is MPEG-2 bit stream. The output of the parser is an interleaved stream of quantized macro blocks encoded in the frequency-domain, and offset encoded motion vectors. In the following steps, the quantized macro blocks are inverse transformed while motion compensation is performed to decode offset encoded motion vectors.
- *Inverse Transformations* step is due to the spatial redundancy reduction at the MPEG-2 encoder. The inverse transformations map each 8x8 block from the frequency domain back to the spatial domain. Each block is reordered, inversely quantized and then followed by an inverse DCT. Similarly, encoded motion vectors are decoded.
- *Motion Compensation* step is due to the temporal redundancy reduction at the MPEG-2 encoder. It performs the motion compensation to recover predictively coded macro blocks. The motion compensation uses the motion vectors to find a corresponding macro block in a previously decoded reference picture. The reference macro-bock is added to the current macro-block to recover the original picture data.

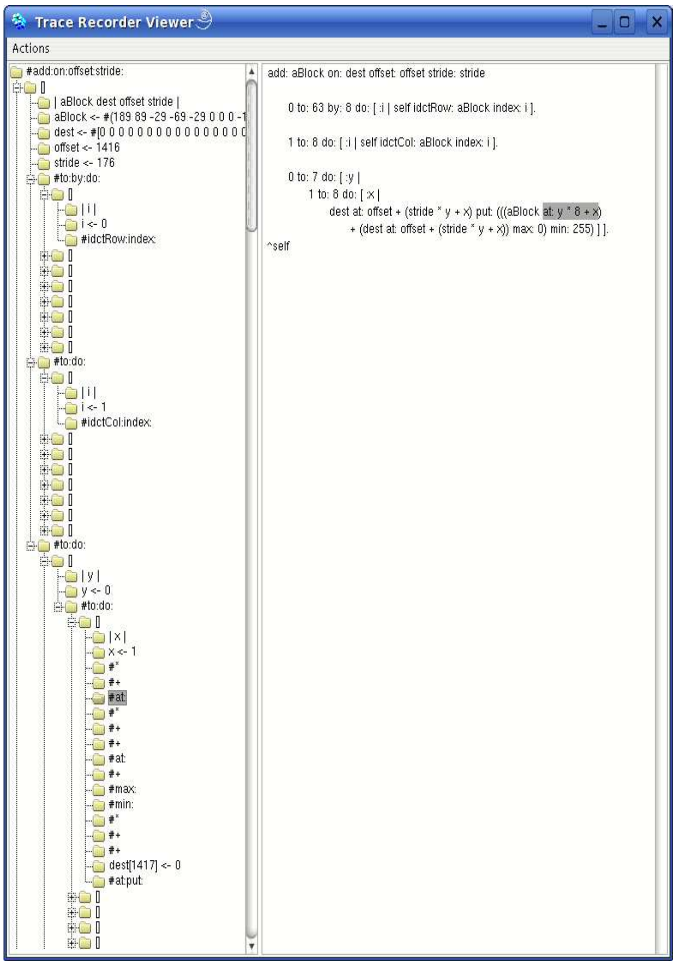
For simplicity, we experiment with 2D Inverse Discrete Cosine Transform and Huffman Decoding to illustrate our analysis approach.

B. Trace Tree Representation

The figure 3 shows the trace tree representation of a single function and its corresponding source specification.

It consists of two parts. The right hand side shows the original source specification (written in Smalltalk) while the left hand side shows its corresponding trace tree representation showing all the operations/events according to the execution order. The right hand side of the figure 3 highlights the portion of the original source code according to the selected operation in the trace tree on the left hand side. It means one can easily go through the trace tree step by step and visualize all the entries (*RecordBlock*, *RecordItem*, *RecordVariable*, *RecordMethod* and *RecordAssignment*) of a trace tree in each step of the source code execution. The symbol # shows a *RecordItem*. However, the symbol # at the top of trace tree shows a *RecordMethod*. Similarly symbol { } shows a *RecordBlock* and so on.

An *Actions* menu regroup three commands:



```
#add:on:offset:stride:
  []
    | aBlock dest offset stride |
    aBlock <- #(189 89 -29 -69 -29 0 0 0 -1
    dest <- #[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    offset <- 1416
    stride <- 176
    #to:by:do:
      []
        | i |
        i <- 0
        #idctRow:index:
      []
      []
      []
      []
      []
      []
      []
    #to:do:
      []
        | i |
        i <- 1
        #idctCol:index:
      []
      []
      []
      []
      []
      []
      []
    #to:do:
      []
        | y |
        y <- 0
        #to:do:
          []
            | x |
            x <- 1
            #*
            #+
            #at:
            #*
            #+
            #+
            #at:
            #+
            #max:
            #min:
            #*
            #+
            #+
            dest[1417] <- 0
            #at:put:
          []
          []
          []
          []
```

```
add: aBlock on: dest offset: offset stride: stride

    0 to: 63 by: 8 do: [ :i | self idctRow: aBlock index: i ].

    1 to: 8 do: [ :i | self idctCol: aBlock index: i ].

    0 to: 7 do: [ :y |
        1 to: 8 do: [ :x |
            dest at: offset + (stride * y + x) put: (((aBlock at: y * 8 + x)
                + (dest at: offset + (stride * y + x)) max: 0) min: 255) ] ].
^self
```

Fig. 3. Trace Tree Representation and Corresponding Source Specification

- *Update* refreshes the trace tree display with the current contents of the trace.
- *Reset* resets the trace recorder, removing all recorded events.
- *Uninstall* uninstalls instrumentation from the instrumented methods.

A trace tree is generated for each function in the source specification of the application using the flow in figure 1 and individual trace trees of each function are combined to get the final trace tree of the application that characterizes the complete application [27].

C. 2D Inverse Discrete Cosine Transform

2D IDCT (for 8x8 image blocks) source specification is first transformed into trace tree using the flow in figure 1. We perform analysis operations on trace tree representation to get analysis results for 2D IDCT. Table I shows the analysis results for different functions in 2D IDCT.

1) Orientation: 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

TABLE I

Orientation Results for 2D IDCT

Function	Computation	Memory	Control	P
idctCol:index:	76.36	23.64	0	1
idctRow:index:	76.36	23.64	0	1
add:on:offset:stride:	77.11	22.89	0	24.14

trees have the same orientation values for both methods as seen in table I.

The first observation is that the percentage of control operations is zero for all the methods, since it is composed of deterministic loops and does not contain any test. Secondly we observe that computation percentage for 2D IDCT functional blocks are higher so it is computation oriented. The results also show a good percentage of memory operations.

Figure 4 shows the percentage of each type of computation in the 2D IDCT. It does not contain any floating point but only integer type operations, i.e. processors with dedicated floating point units are not necessary and processor selection should focus on integer performance instead. Furthermore, 27 % operations are multiplications, so selected processors may have dedicated hardware multipliers.

The fact that there is no need for complex control structures, the high data-accesses requirements and the coarse grain parallelism mean that optimizations can be obtained with a pipelined architecture with possible coarse grain dedicated hardware modules providing a large bandwidth. So if high performances are required, an ASIP or a programmable dedicated hardware can be introduced within the SOC.

2) Spatial Parallelism: 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 (methods) are sequential in nature and does not contain any inherited parallelism. However the level of parallelism increases at the higher level of granularity (2D IDCT). The value of P at this level is 24.14 indicating that a coarse grain parallelism is available.

D. Huffman Decoding

We first generate the trace tree for each function of Huffman decoding and then combine the individual trace trees to get the complete trace tree of Huffman decoding using the flow in figure 1. We perform analysis operations on trace tree representation to get analysis results. Table II shows the analysis results for representative functions of Huffman Decoding.

1) Orientation: It can be noticed that these functions have relatively high percentages of control operations denoting heavily conditioned data-flows. 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 methods are control and computation oriented. Figure 4 shows the percentage of each type of computation in the Huffman

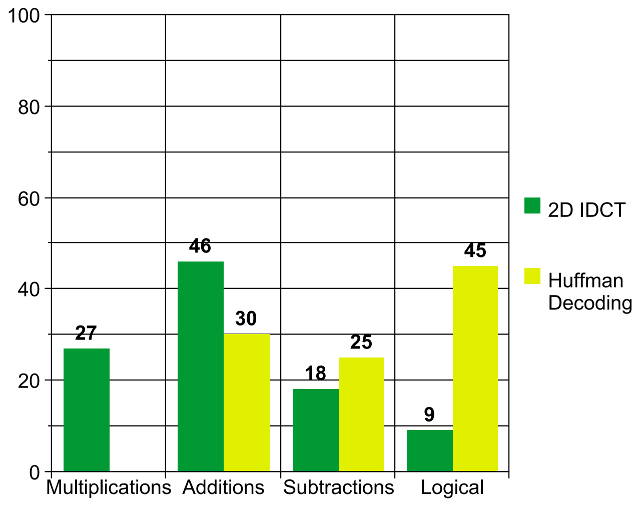


Fig. 4. Percentages Of Computation Types For 2D IDCT and Huffman Decoding

TABLE II

Orientation Results for Huffman Decoding

Function	Computation	Memory	Control
getChromaDCDctDiff	49	2	49
getCodedBlockPattern	52	5	43
getLumaDCDctDiff	60	2	38
getMacroblockAddrIncrement	50	5	45
getMacroblockMode:	58.3	8.4	33.3
getMotionDelta:	58.2	3	38.8
getQuantizerScale:	75	0	25
Huffman Decoding	60	7	33

decoding. There are no floating-point but only integer-type operations. Furthermore, there are no multiplications, so selected processors have no need for dedicated hardware multipliers. The results show that 45 % of the computations are logical operations.

2) Spatial Parallelism: We have not shown the value of P in table II because the value of P remains 1 at all hierarchical levels of trace tree. It reveals that suitable target architecture for Huffman decoding algorithm may be a GPP (General Purpose Processor). There is no need for a DSP and for a complex data path structure, since the parallelism cannot be exploited at any level.

VI. Conclusions

In this paper we have proposed a high-level application analysis methodology which aims at guiding the embedded systems design process for early design space exploration targeting the multimedia domain. Application is specified in a higher level object oriented language (Smalltalk) and then transformed into a trace tree representation by dynamic analysis. Unlike conventional dynamic analysis techniques, instrumentation is done on abstract syntax tree rather than source code. Instrumented application is then executed to get trace tree representation. Due to the diverse requirements of embedded system design space exploration, the proposed framework enables the characterization of applications trace tree by a generic analysis approach which is not restricted to only a set of metrics. Designer can extend the framework by defining new operations on the trace tree of the source specification. To illustrate the methodology, we discuss the significance of some analysis results in the design space exploration of embedded systems and perform analysis operations on trace tree representation of the application. The outcome is a set of guidelines characterizing the application in terms of processing, control and memory orientation as well as in terms of potential spatial parallelism. Experiments with 2D IDCT and Huffman decoding algorithms shows that our approach not only helps designers to intensively comprehend the application but also provides valuable indications to highlight architectural opportunities and directions to improve application architecture matching.

The proposed framework is a starting point of the complete design flow and is still in its early phase of development. Using the proposed methodology, we are developing a framework for automatic conversion of sequential programs to parallel programs dedicated to streaming applications. We are working to integrate our analysis framework with the available synthesis tools by scheduling the trace tree to a family of coarse grained reconfigurable architectures.

References

[1] H. J. Stolberg, S. Moch, L. Friebe, A. Dehnhardt, M. B. Kulaczewski, M. Berekovic, and P. Pirsch, "An soc with two multimedia DSPs and a RISC core for video compression applications," *Digest of Technical Papers. ISSCC.*, pp. 330–531, Feb. 2004.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," vol. 25, no. 2, Mar./Apr. 2005, pp. 21–29.

[3] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2005, pp. 258–262.

[4] W. Eatherton, "The push of network processing to the top of the pyramid," in *Symposium on Architectures for Networking and Communications Systems*, Princeton, New Jersey, USA, 2005.

[5] P. M. Kuhn and K. P. M., *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[6] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, 2004.

[7] H. Zuse, *Software complexity: measures and methods*. Hawthorne, NJ, USA: Walter de Gruyter & Co., 1991.

[8] I. Tomek, Feb, 2002. [Online]. Available: http://www.iam.unibe.ch/ ducasse/FreeBooks/Joy/

[9] "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.

[10] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*. New York, USA: ACM Press, 1995, pp. 456–461.

[11] P. Puschner and C. Koza, "Calculating the maximum, execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[12] S. Mallat and F. Falzon, "Analysis of low bit rate image transform coding," *Signal Processing, IEEE Transactions on Speech, and Signal Processing*, vol. 46, no. 4, pp. 1027–1042, 1998.

[13] T. Ball, "The concept of dynamic analysis," in *Foundations of Software Engineering*. Los Alamitos CA: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, 1999, pp. 216 – 234.

[14] A. Lienhard, S. Ducasse, T. Grba1, and O. Nierstrasz1, "Capturing how objects flow at runtime," in *Proceedings of the 2nd Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*. IEEE, 2006, pp. 45–49.

[15] M. Denker, O. Greevy, and M. Lanza, "Higher abstractions for dynamic analysis," in *Proceedings of the 2nd Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*. IEEE, 2006.

[16] A. Hamou-Lhadj, "The concept of trace summarization," in *Proceedings of the Ist Workshop on Program Comprehension through Dynamic Analysis (PCODA'05)*. IEEE, 2005, pp. 43–47.

[17] J. Brant, B. Foote, R. E. Johnson, and D. Roberts, "Wrappers to the rescue," in *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 1998, pp. 396–417.

[18] M. Ravasi and M. Mattavelli, "High abstraction level complexity analysis and memory architecture simulations of multimedia algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 673–684, May 2005.

[19] L. Cai, A. Gerstlauer, and D. Gajski, "Multi-metric and multi-entity characterization of applications for early system design exploration," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, New York, USA, 2005, pp. 944–947.

[20] F. Salice, L. D. Vecchio, L. Pomante, and W. Fornaciari, "Partitioning of embedded applications onto heterogeneous multiprocessor architectures," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. New York, USA: ACM Press, 2003, pp. 661–665.

[21] Y. Moullec, J.-P. Diguet, N. B. Amor, T. Gourdeaux, and J.-L. Philippe, "Algorithmic-level specification and characterization of embedded multimedia applications with design trotter," *J. VLSI Signal Process. Syst.*, vol. 42, no. 2, pp. 185–208, 2006.

[22] J. Reichel and M. J. Nadenau, "How to measure arithmetic complexity of compression algorithms: asimple solution," in *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, vol. 3, New York, USA, 2000, pp. 1743–1746.

[23] S. Rul, H. Vandierendonck, and K. D. Bosschere, "Function level parallelism driven by data dependencies," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 55–62, 2007.

[24] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl, "Sprint: A tool to generate concurrent transaction-level models from sequential code," *EURASIP Journal on Advances in Signal Processing*, pp. Article ID 75 373, 15 pages, 2007, doi:10.1155/2007/75373.

[25] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 151–162, 2006.

[26] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, Apr. 2006.

[27] T. Goubier, "The Trace and Dynamic Program Analysis Framework," Architectures et Systmes, UBO, Brest, Tech. Rep., May 2007.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley Professional, January 1995. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20&path=ASIN/0201633612

[29] L. L. Hours, "Etude et Modlisation MPEG 2," Architectures et Systmes, UBO, Brest, Tech. Rep., 2002.