GAIA: A GENERAL APPLICATION INSTRUCTION SET and ARCHITECTURE EXPLORER

by

Ayşe Gaye Soykök

BSc, in Computer Engineering, Istanbul Technical University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2008

ACKNOWLEDGEMENTS

First, I would like to thank my thesis supervisor Arda Yurdakul for her insight into the research conducted, her motivational support and for her patience for the times I needed a break to sort the work out.

I also would like to thank Can Özturan and Günhan Dündar for their patience and feedback and their willingness to spend time being examiners of my thesis.

I thank people in Boğaziçi University offering such a wide range of classes that I could find many of the topics I am interested and curious in, giving an insight one by one.

I thanks my friends Hilal Akay and Pınar Karagülle for their help on this thesis to sort some parts out. I think they deserve a big credit for being so motivational and being so smart. I thank my dearest friend Neşe Alyüz and Necmiye Genç for helping me get hold of the thesis text style. I thank Demet Nar for being there. I am so happy to even get to know you ladies. And I thank Neşe for carrying my precious guitar all way long.

I thank my family for their patience in the times I was not even reachable, for their support and trust. And finally for the values they have taught me.

I thank also to the unknown "bus driver" who will never know he created such a basis, for his sincere, unexpected wish making me hold on with the words when things did not go so smooth.

This thesis has been supported in part by TUBITAK, the Turkish Foundation of Science and Technology, under Kariyer Research Project Program, under project number 104E038.

ABSTRACT

GAIA: A GENERAL APPLICATION INSTRUCTION SET and ARCHITECTURE EXPLORER

Embedded Systems are dedicated to a task for their life time with no or slight modifications. These systems are necessary in a wide range of industrial areas from entertainment industry to cryptography and from house appliances to army equipment. The emerging of processors with customizable instruction sets and customizable architectures has enabled the embedded processors to be tailored for the application they are dedicated to. Tailoring stands for improving incompetent parts of an application by modifying the processor.

Development of design automation tools have been a new research era for embedded processors. They enable customization either by partial automation which requires human assistance at varying levels or by full automation.

In this thesis, an automation tool GAIA that selects custom instructions (CI) and Single Instruction Multiple Data (SIMD) style processing elements (PEs) has been developed. The system achieves customization by examining the intermediate representation (IR) of an application. It is a fuzzy expert system that acts as a voting mechanism evaluating the attributes of the application components. The work of this thesis contributes to the stage between the front-end and back-end compilation, with the aim of assisting back-end compilation at customization process.

ÖZET

GAIA: GENEL UYGULAMALAR İÇİN KOMUT SETİ VE MİMARİ ARAŞTIRICISI

Gömülü Sistemler çalışma süreleri boyunca bir uygulamaya, değişiklik olmadan ya da ufak değişiklerle adanmış sistemlerdir. Bu sistemler, eğlence sektöründen kriptografiye, ev eşyalarından askeri ekipmana kadar geniş bir sektör yelpazesine hizmet vermektedir. İşlemcilerin uyarlanabilir komut setlerine sahip ve mimarilerinin yapılandırılabilir olması ile birlikte, gömülü sistemlerin atandıkları uygulamaya göre yapılandırılma esnekliği sağlanmıştır. Yapılandırma, uygulamanın yetersiz olan bir kısmını iyileştirmek için, işlemcide değişikliğe gidilmesi anlamına gelmektedir.

Kullanıcının çeşitli düzeylerde müdahelesini gerektiren yarı otomasyon ya da tam otomasyon ile yapılandırma sağlayan araçlar, gömülü sistemler sektöründe yeni bir araştırma dalı ortaya çıkarmıştır.

Bu tez, yapılandırılabilir komut ya da Tek Komut, Çoğul Veriyolu (TKÇV) stili işlemci elemanlarını, gömülü sistemin ara gösterimini inceleyerek seçen bir araç sunmaktadır. Bulanık uzman sistemi, bir oylama mekanizması olarak kullanılır. Bu tez, ön derleme ve son derleme safhalarının ortasında yer alarak, yapılandırmada son derleme safhasına destek olacak şekilde tasarlanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
CI	Custom Instruction
CDFG	Control Data Flow Graph
CRA	Coarse Reconfigurable Architecture
COG	Center of Gravity
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DMA	Direct Memory Access
GES	GAIA Expert System
GGE	GAIA Graph Evaluater
GPP	General Purpose Processor
HDL	Hardware Description Language
IP	Intellectual Property
ISA	Instruction Set Architecture
LHS	Left Hand Side
OS	Operating Systems
MIMD	Multiple Instruction Multiple Data
RHS	Right Hand Side
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
VHDL	Vhsic Hardware Description Language
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time
PE	Processing Element

1. INTRODUCTION

In demand of faster system requirements in computer industry and with the applications getting more complex and compute intensive, smaller clock frequencies and deeper pipelines have been implemented for General Purpose Processors(GPPs) in order to speedup processors. Embedded systems, having strict and demanding specifications to meet, necessitated a totally different approach. The approach taken in the area of embedded systems is the customization of the processor to the application or to the domain of the application.

Customization is not used extensively in GPPs because customization necessitates a whole software tool chain for the instruction set variations. Also it introduces incompatibility of platforms for applications. On the other hand, embedded processors are generally dedicated to a specific task and execute it for their overall life time with slight or no modifications. Cipher coder/decoders, image processors, audio processors can be given as an example of embedded systems. The common point for these applications is that they have a computation intensive part that can improve if treated in a specific way. Tailoring the processor according to the needs of an application is called customization. The application statistics can benefit by means of several properties such as overall execution time, critic-path execution time, area, response time, power consumption etc. Among these properties overall execution time is the most generic one to delve into.

An application can be treated specifically by means of hardware/software (HW/SW) codesign and/or parallelization. In instruction customization concept, HW/SW codesign term means migrating the instruction or a set of instructions to the hardware. Parallelization of tasks is possible by exploiting data independent loops consuming a high ratio of the execution time in applications. Tailoring the architecture and the instruction set, has been a possible approach with the introduction of Application Specific Instruction Set Processors (ASIPs) and soft processors.

1.1. Terminology

Before proceeding further, terminology used in customizable processors is presented in this section for clarification. Processor "customization" may stand for both the architecture and the ISA(Instruction Set Architecture)of a processor. ASIPs are processors that come with a fixed architecture but extensible instruction set, where it is possible to "customize instruction set" and synthesize customized instructions to a programmable hardware logic. "Architecture customization" is possible in soft processors where generally a CPU core template with modifiable architectural features exists in an Hardware Description language(HDL).

The abbreviation for processing element (PE), stands for concurrently executing several processing elements which communicate with GPP as well as among other PEs, whereas (custom instruction) CI is a single element that resides on the datapath of the GPP.

SISD+CI is a processor architecture with a Central Processing Unit(CPU) core coupled with a set of CIs. Processor and CIs execute sequentially where CIs are integrated to processor's datapath. SIMD architecture stands for concurrently executing PEs that are identical. Processing elements reside along with GPP. When VLIW term is mentioned, concurrently executing non-identical PEs along with GPP is meant. In some terminologies, VLIW stands only for instruction cascading in a longer instruction and parallel PEs can be identical.

1.2. Automation in Customization of Embedded Systems

Automation in customization of embedded systems is an emerging branch in customizable embedded systems area. After the advent of customizable processors, the knowledge in the domain had to be acquired and applied to the application under evaluation. Architecture customization is conducted with the help of user interaction. An expert choose the parts that need special attention [1]. A tool may help the user to mark the problematic parts of the application and specify what and how it is to be

done. A further step is to automate the process and leave the customization process to the tool. The recent research and challenge are in this area [2], [3], [4], [5], [6].

The stages in the customization of embedded processors is given in Figure 1.1, for an overall insight. The schema stands for a general approach. High level design part enables user to specify the embedded application with a high level representation such as C, UML or a graphic language. Hardware library and constraints for the system are extracted from the user, along with the application software. Hardware library consists of IP cores, specification of the IP cores in terms of clock cycle, power consumption, HDL description, area etc. System constraints can be the area the system must fit, IO specifications, the frequency system must operate on etc. The high level stage provides input to the intermediate design part where partitioning; CI/PE selection takes place. Inputs to the decision phase are the system specifications, HW library and the intermediate representation of the application for control and data flow. The decision part is the core of the automation of customization, where CI/PE exploration is performed. Data flow graph (DFG) to HDL converter stands for the HDL construction module for the HW partitions marked in the intermediate representation. After decision phase comes the realization phase of the system. HW partitions are synthesized into a configurable logic. Custom instructions' opcodes are added to the instruction set of the processor and the whole application is compiled into assembly code.

This thesis focuses on automating the instruction selection and identification process and the way instruction customization will be treated.

In the following subsections, details on instruction set customization, architectures in customizable processors are given. The last subsection is about fuzzy logic and expert systems on HW/SW codesign literature.

1.2.1. Instruction Set Customization

The idea behind migrating from General Purpose Processors (GPPs) to Application Specific Instruction Processors (ASIPs) is to customize the instruction set of the

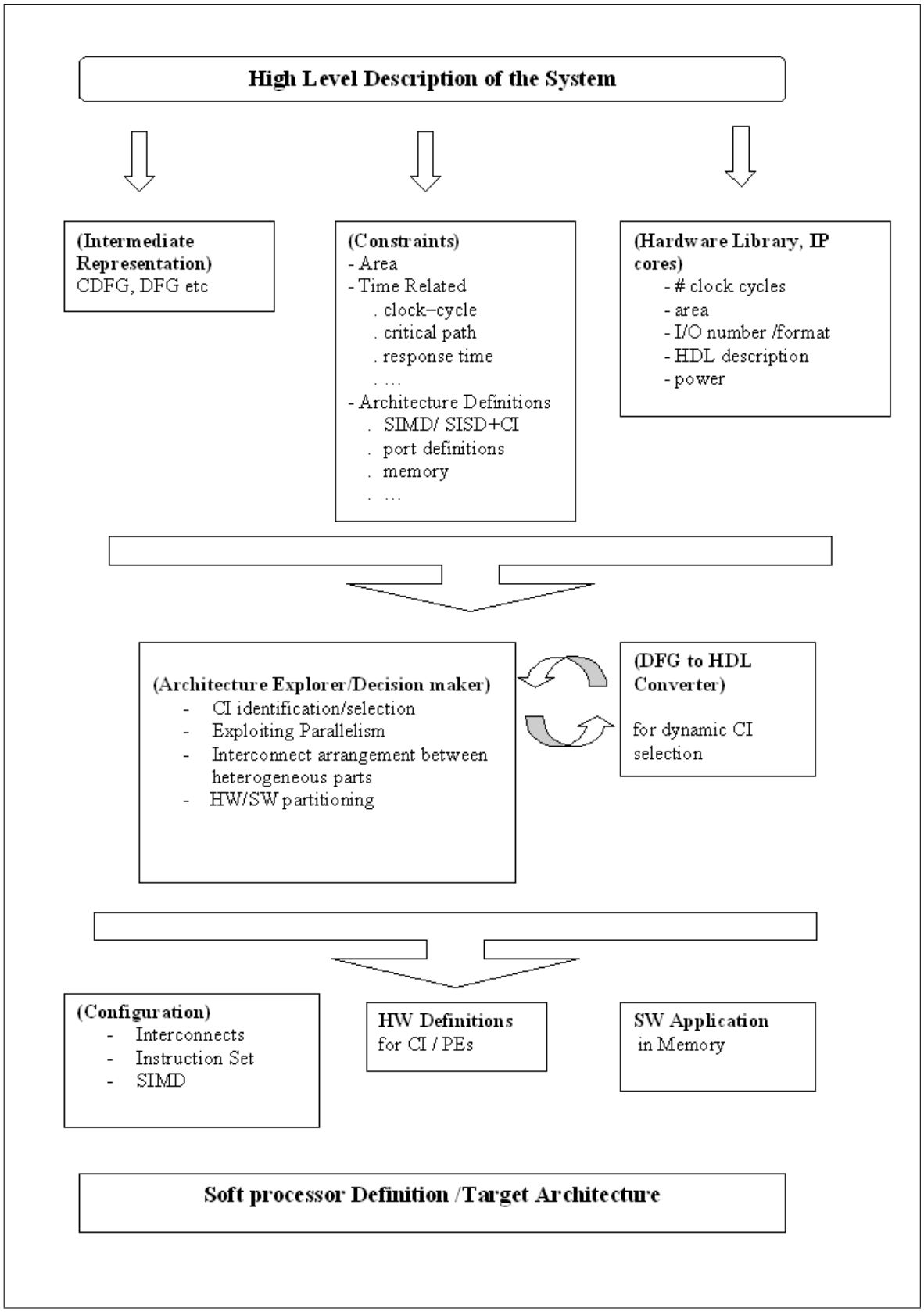

Figure 1.1. Processor customization overview schema

processors, depending on the application (or the domain of the application [7]) so that it merits either average or worst case execution time.

When there is no automation, a set of tools are provided so that it enables the expert user to develop a new instruction set [1]. The decision is based on user's experience and knowledge. This requires a team of people with adequate knowledge of HW and SW.

Automating customization engine, eliminating user interaction to some extent is the final approach. Profiling of the application, detection of hot spots and the instructions that increases efficiency in any means is left to the tool.

Generally the input to the instruction set customization is a graph representation of the application, although it is not the only approach. The graph models vary for the control and data flow of the application. Control flow graph(CFG) represents the control statements of the application like *loops*, *IF* statements, branches etc. Data dependency is modeled in a data flow graph(DFG). Generally CFG and DFG is blended into one graph; control data flow graph (CDFG). CDFGs with different variations are used. Some of the examples are hierarchical CDFG [8] or acyclic CDFG [9]. A survey of graphs and ASIPs, can be read in [10].

Instruction set extension can be classified under two topics;

- Instruction set generation for which possible custom instructions are chosen (section 1.2.1.1).
- Selection of custom instructions that will implemented in customizable logic, from a set of feasible custom instructions (section 1.2.1.2).

1.2.1.1. Instruction Identification. Identification of custom instructions means extracting patterns from the intermediate representation of the system application. An example is is enumerating patterns in a DFG which are feasible to become custom instruc-

tions [11]. Some of the work in literature targets just identification and optimization of identification process excluding the selection [3], [12]. All patterns are not feasible for ASIPs because of micro-architectural constraints such as input/output sizes due to register ports, area or forbidden tasks as memory operations. The work in literature has been either carried out to overcome these restrictions to generate more efficient custom interactions [13], [14] or driven by the restrictions employing them as heuristics to reduce search space [2], [3], [11], [12].

An approach that profiles the application to vote for more beneficial blocks to be chosen as CI has been carried out, not evaluating the feasible custom instructions in the optimization function [15]. It uses enumeration method in [11] This is inline with our idea that the attributes of elements have to be taken into account. Considering just the topology of the graph, in other words edges connecting the nodes, without evaluating the attributes of the nodes is not sufficient for specialization of customization.

There are works that eliminate architectural constraints from identification process. Work carried out in [7] employs a recurring pattern identification method to create a library of patterns. This work has a similar approach with this thesis in the idea of targeting recurring patterns. It differs in the way it is used for creating a pattern library, for several applications in one domain. Recurring pattern identification stands for finding isomorphic subgraphs that exist several times in a graph. It is also called as recurring subgraph problem [16]. An earlier work that pre-existing library templates such as multiply-accumulate and shift-add chains [17]. It searches for the templates in the application, and marks them as custom instructions. Another recurrent subgraph approach has been used in custom instruction identification area is mentioned in [1]. The algorithm called *sequences* detects only recurrent sequences of two instructions such as multiply-accumulate and shift-add.

The following sections summarize the restrictions that are evaluated in literature in instruction selection phase.

Input/Output size

The maximum number of input output operands depends on register files and feasibility of instructional encoding. However it is an artificial restriction that is imposed by the designer to prune design space. The smallest input/output pattern is 2/1 input/output. MISO stands for multiple inputs but a single output. K inputs may exist in the instruction but there is a single output. MIMO is the acronym for multiple-input multiple-output.

The number of operands is highly related to the speedup gained by implementing custom instructions. It has been figured out 2 to 4 inputs perform with reasonable speedup [2].

Memory restriction

Memory is one of the restrictions evaluated [3]. Patterns that include memory operations are left out of the set of feasible patterns for instruction generation, in order not to involve any memory access instruction inside custom instruction pattern. Memory load operations are an example. This restriction arises because the system under design has a fixed architecture to an extent and size of the register file cannot be changed. Since soft processors do not have hard architectural constraints, it is possible to eliminate memory restriction from custom instruction identification stage. For soft processors, distributed memory may be created along with instructions.

Block boundaries

One other aspect mentioned in instruction generation is implementation of instructions that cross basic block boundaries. Basic blocks in a CDFG are DFG partitions that does not contain any control nodes. Crossing block boundaries introduces IO size increase in CIs generally, although there are exceptions such as the ADPCM decoder in [18]. If the patterns are chosen cleverly, there may be a reduction in temporary registers to pass data between blocks. The area required for an instruction will

increase, resulting fewer instructions to be chosen due to area. It is mentioned crossing block boundaries do not outperform [3]

Number of custom instructions

As the number of custom instructions increases, the performance of the system rises increases in a nonlinear fashion. However it is not possible to implement everything in custom instructions. One reason is the area constraint of the system. The other is the number of instructions imposed by the decodable opcode range in the processor. A research on the number of custom instructions is given in [2].

1.2.1.2. Instruction Selection. The second phase of instruction extension design is instruction selection. Some of the feasible custom instruction patterns do not contribute to performance optimization of the system. Since there are limited number of resources there is the necessity to choose optimal or suboptimal solutions. The problem is also modeled in operational research and economics. There are restricted resources and an optimization function. The task is to find a solution that maximizes the benefit. Integer linear programming(ILP) based methods inherited from operational research are used extensively for the solution of the problem. The following sections summarizes the factors evaluated in optimization function.

Exploiting critical parts of the application

Most of the applications aim to reduce average case execution time (ACET) which results in faster systems. For real time systems, there are timing constraints which can be *hard*. *Hard* phrase means that a specific task must be done in a specified span of time. In such a case, the aim should be speeding up critical paths, which perform poorer than they ought to be. Worst case execution time (WCET) reduction has been mentioned in literature. The work carried out in [4] reduces the WCET of an application, to serve real-time embedded applications requirements. Timing values of a system are constructed from its CDFG by traversing it. The custom instructions

are chosen from paths marked as possessing the worst case execution time.

WCET reduction idea lacks the fact that the path performing the worst may not have a hard constraint. A path in WCET range may have a soft time constraint which results in some slack to finish the required task. In this case, there might be a faster path with a hard constraint and the path may not be satisfying this timing requirement imposed. If the customization is performed for real time systems, the criticality of the timing constraint should also be introduced into the problem.

Exploiting average execution time

Instead of healing the paths with worst time consumption, the aim is to heal the overall performance in terms of system speedup. A path with WCET may not be on a common executing path, so healing WCET would not contribute to the performance optimization of the overall system. The common approach in literature is maintaining an overall speedup, reducing average case execution time (ACET) [2].

The enumerated CI patterns that are most common and most hit in the execution of a program are chosen in a DFG, so as to optimize average execution time. Loops constitute an example for the CI patterns that are hit most, as in general HW/SW partitioning problem.

1.2.2. Exploiting Parallelism

Parallelism in an application may exist as data, instruction, and pipeline parallelism. Data parallelism is exploited by SIMD, instruction parallelism by VLIW and pipeline parallelism by instruction fusion.

It is a fact that much of the execution time is consumed by loops in an application. Migrating the loops or part of the loops to hardware is beneficial. Parallelization of loops in an appropriate way is considered to be more advantageous in terms of execution time [1]. As an example of data parallelism the below loop can be considered.

```
For i = 0 to i = k
    Task A[i]
```

If task A[i] is independent of A[i-1], task A can be executed in parallel. Whenever task A instruction is detected in fetch decode cycle, N similar PEs, executing k/N times is called SIMD style parallelization. Here the granularity of PEs may differ. PEs may be fine grained and data input-output can be handled via register file or they might be having their own local memory. This way of processing is called heteregenous multiprocessing in a source [1]. Note that in literature, SIMD term in "SIMD instructions" does not necessarily mean instructions are the same for all PEs. VLIW style architectures are also referred [19], [5] with the same term. In VLIW style, PEs do not have to be the same. Instructions for PEs are packed into one long instruction and executed in parallel. The below loop constitute an example for parallelization and loop regulation.

```
For i =0 to i = k
   Task A[i]
   Task B[i]
   Task C[i]
```

Tasks A, B and C can be executed in different PEs in parallel if they are not data dependent on each other. Even if they are data dependent, regulating the instructions in loops, which is called software pipelining, makes the parallel execution possible [20]. Software pipelining is an approach used together with loop unrolling. It is used for handling data dependencies in loops. Pipelining does not occur in the architecture but it is an arrangement of instructions in the loop [1]. Data belonging to the previous indexed loop executions are prepared prior to execution of latter indexed ones.

In parallel processing, arranging communication in PEs is a term to consider. The speed of execution and efficiency of communication depend on the method used. The soft processor implementation carried out in [21] makes it possible to arrange the

interconnects via a look-up table.

1.2.3. Architectures in Customizable Processors

The given specification may be mapped onto a SISD+CI, SIMD, VLIW or MIMD architecture with an appropriate processor core. The choice may be specified by the user, dictated by the underlying target hardware or chosen by the tool. One soft processor can support several of the architectures.

1.2.3.1. SISD+CI. SISD+CI architectures have a SISD style GPP coupled with CIs. The instruction set of the GPP is cutomized and additional CIs are added to the set. There can be several CIs added to instructon set. However, execution of CIs are sequential and one at a time. The CIs reside in the datapath of the GPP which has the control. CIs are functional after a fetch decode cycle. GPP also executes sequentially with the CI set.

1.2.3.2. SIMD. A branch of the literature is dedicated to SIMD Processor cores [5],[19]. VLIW architectures can be found and phrased as SIMD cores also. With an assembly code and timing constraints, the code is constructed as SIMD operations [5]. The hardware is then mapped to a RISC or DSP kernel with SIMD functional units. SIMD functional units are duplicated around the processor core. The custom instructions are SIMD type.

1.2.3.3. VLIW. VLIW architectures stand for concurrently executing PEs residing along with the GPP, where PEs are not identical. VLIW stands for the different instruction commands cascaded together in one long instruction. Each sub-instruction is decoded for one PE. From one perspective, if the sub-instructions are identical, PEs can also be identical. There are works carried out in literature specialized for VLIW architectures [5], [22]. Applications with vector operations or applications that can exploit instruction level parallelism are beneficial to execute on VLIW architectures.

1.2.3.4. Coarse Grained Architectures. In spite of not excluding the classification regarding data/instruction attributes in the above sections, another architecture type is coarse-grained style. CRAs (Coarse Grained Reconfigurable Architectures) are targeted in [23] with a clustering method. The method is applied to loop specifications of the program.

1.2.4. Terms that Benefit from Customization

Architecture and Instruction set Customization is performed so that particular embedded application or a domain of applications is improved in an aspect. Generally improvement is achieved in time related issues considering area and power constraints.

Overall execution time is the most profitable and worked on issue. Overall execution time can be achieved with no additional constraint in the CI identification process. The aim is to maximize speedup via loop parallelization, implementation of HW versions for complex and special functions, migration of time consuming and recurring patterns to HW as CIs.

Some applications for real time systems have hard timing constraints for some specific tasks. Instead of overall execution, speedup in specific partitions may be more mandatory. Considering application specific architectures, attention should be on user requirements and inputs. Hard constraints on timely issues constitute an example for application specialization, which is a must for real time systems.

1.2.5. Challenges in Customization

Below section explains the terms mentioned in customization literature, both from PE and CI point of view. The terms belong to constraints to meet or consequences to consider.

Area

Total area required for the system increases with the introduction of custom instructions. However, improvement on timing is not possible without migration of instructions to hardware. Area increases because the CIs or PEs are migrated to hardware which consumes additional space other than the processor core. Increase in area is higher with parallelization of tasks via PE usage. During design space exploration, area increase should be included in the model and must be pre-estimated so that a feasible solution is found in a reasonable time span. There may be parallelism in the application that is beneficial to exploit however because of the area budger, a solution with SISD+CI may be chosen which generally consumes smaller area compared to PE solution.

Memory

Memory is another constraint for architecture customization, although it is not as critic as area. Program data memory may increase during the introduction of custom instructions. Custom instructions and PEs can have their own local memory blocks since memory operands that require access to memory are not restricted. This results in distributed memory throughout the softprocessor.

Increase in memory size is also possible because of methods like loop unrolling and software pipelining. Recurring instructions are expanded or regulated, and written into memory. On the other hand, instruction fusion may decrease code memory because several instructions can be indicated by only one instruction.

Memory has been one of the design limitations in the litarature [3], [11]. Search space is also reduced by removing memory-related operations. Forbidden nodes are the nodes in the DFG that include memory access and excluded from CI identification process [3], [11]. However, the real benefit may be in the CIs/PEs that include memory access. Loop or vector operations that access and process data in some specific partition of memory can be taken as an example. The data memory can be divided into chunks

and be processed in parallel. PEs executing in parallel have their own local memory . This SIMD style processing is called heterogenous multiprocessing [1].

An approach has been considered to overcome the exclusion of memory potential problem [24]. A work in soft processors is observed in [21] to include PEs with local memories. The availability of local memory in PEs directs the literature to include operands with memory access while pruning design space. Although introducing memory access operands increases the time overhead to get PEs operational, it increases speedup efficiency for data independent and data-intensive applications.

Introducing memory operands requires an estimation for the size of a caching module to be merged to the system which will copy a part of the global memory to the local memories for parametric access. A block of potentially accessible memory is downloaded to local memories prior to execution . This results in removing the necessity to access to global memory during execution of CIs or PEs and enabling them to execute atomically.

Input/Output

Custom Instructions transfer the required input/output (I/O) data through register files. I/O constitutes a limit for the solution space due to register file's size. Register files are fixed structures in ASIPs, although there is the possibility to change them in soft processors. A branch of the literature has been shaped by the fixed size of register files limiting/parameterizing I/O number of CIs [11], [3].

Complexity of the search space

CI selection and identification process can take huge execution time depending on the optimality of the algorithm and pre-assumed specifications. In [11] the enumeration method is called slow because it evaluates all the DFG of the application in order to derive CIs from it. The method proposed in [3] is faster. It relies on some heuristics and does not explore the whole design space. As a result it is not optimal. The

introduction of forbidden nodes which are memory accesses and I/O constraints speeds up the algorithm, decreasing search space. The harder the constraints are for the input output specification, the faster the algorithm is due to the search space collapsing.

The disconnectivity of the graphs also affects the complexity of the search space. Disconnected DFGs can exploit parallelism in nature. However spanning disconnected graphs increases time complexity of the algorithm employed. Infeasibility of search space has been reported in [12], [3].

Clock Cycle

The clock frequency of the system is an imposed constraint of the system by the user. Whenever CIs/PEs are created, clock frequency constraint may be violated. The communication overhead introduced by the coupling of CI/PE with the main processor or the latency of the combinational logic in the PEs or CIs constitute a challenge for achieving required clock freguency. It may not be possible to implement large combinational PEs for high clock frequencies. The latency of the CIs/PEs should be estimated during design space exploration in order not to end up with an infeasible solution.

1.2.6. History of Fuzzy logic and Expert Systems in Hardware Software Codesign

Before ASIPs and custom instructions coming into scene, expert systems have been employed in HW/SW codesign research. The work is, however, in a larger scale where the components are coarse grained functions of an embedded system. The target system consists of a microprocessor coupled with an FPGA or several FPGAs. An approach utilized a fuzzy expert system as the hardware software partitioner, where the instantiated rules are at a larger scale. The rules evaluate line of codes of software, control elements in a module or combination of operations in a module. The resultant component is mapped to an FPGA if it is chosen to be in hardware [25].

In another work carried out for microprocessor based systems, the expert system is employed for knowledge representation for the overall embedded system design process [26]. A part of the expert system performs task partitioning between HW and SW. The scale is large; HW/SW codesign process targeting a whole embedded system as in the previous work mentioned in [25]. Expert system rules considered for partitioning is not mentioned and fuzzy logic is not used in [26]. Another approach in HW/SW partitioning literature, designed for Application Spesific Integrated Circuits(ASICs) employs an expert system [27]. Fuzzy logic is used to model the imprecise terms of HW/SW partitioning phase. It operates on large scale of blocks.

1.3. Motivation and Approach

GAIA is developed for a soft processor architecture. The architecture of a soft processor is not fixed and can be guided by the characteristics of the application or the domain of the application. The design can be free from architectural constraints that are considered by some of the earlier works in literature [3], [11], [12], because the architecture is not fixed. It is obvious that some flexibility is present during instruction identification process. There are some drawbacks introduced. The search space cannot be collapsed without memory and I/O constraints. On the other hand, abstracting the architecture considerations enables the customization of the architecture of the system. The result can be integrated for a soft processor. A back-end and front-end compiler is required to construct a whole tool chain. The resulting tool chain can be used to design a customized embedded system not mandated by the architecture limitations of a specific hardware.

Embedded systems having a broad range of applications and systems, necessitates special handling of the customization process, according to the system and application specifications. GAIA serves different demands of customization imposed by the nature the applications. Serving a broader range of applications is required for a general automation tool in customization. The system specifications, like time critic or area critic, or any special system specifications like reducing a specific path for a real time application that has a hard constraint can be added to the voting mechanism of GAIA

expert system. The work aims to assess conflicting/cooperating attributes of an application and tailor the automation process as well as architecture. Discrete rules vote together for the parts to be implemented in hardware.

Much of the work in literature is carried out to overcome the restriction of an existing architecture. For example, to overcome register file limitation, shadow registers have been implemented so that I/O size of the instructions is increased [13].

Automation in customization research is to "mimic the choices of an expert" [1]. Earlier works have aimed at optimizing some aspect of the CI selection or identification. The work carried out differs mainly in the idea to introduce a broader range of aspects. Customization of the automation process is targeted with hardly optimum methods, as a human expert is likely to perform. An expert system is constructed by extracting *the rules of thumb knowledge* of the domain. The system decides upon the components of the application to be customized according to the attributes of the application and system specifications.

This work resides between the front-end and back-end compiler. To be more specific, it is a component that can be plugged into the compilation process. But it would not be possible to cascade GAIA with front-end and back-end compiler end to end, since it does not carry all the information required in back end compilation such as inputs like register information, data wordlength, memory design, etc. GAIA profiles abstract data and control characteristics of the application and proposes the parts that should be in hardware as a CI or PE.

There are neither architectural constraints nor specifications. The data are abstracted. Therefore the operator granularity is flexible can be as granular as assembly level or as coarse as functional level. A coarse grained architecture can be targeted with a soft processor. For this reason, GAIA should not reside after back-end compiler since the operator granularity would have already been decomposed into instructions. In other words, coarse grained user function specifications are lost after back-end compilation.

1.4. Contribution of Thesis

In customization literature, the ongoing effort targets full automation, *mimicking* the choices of an expert. Tool should exhibit generality and should be able to assist the customization process with *rules of thumbs* of the research area. There has not been any work in instruction customization, to our knowledge, employed as a knowledge database, namely an expert system. In this thesis an expert system which is integrated with fuzzy logic in order to represent imprecise terms of the domain has been implemented. Due to data abstraction, the system is designed to assist back-end compilation process, residing between front-end and back-end compiler.

1.5. Outline of Thesis

The thesis is organized as follows: Chapter 2 describes the GAIA Explorer system in detail in two parts: the Fuzzy Expert system part and the Graph Evaluator part. Fuzzy Expert system basics along with the utilization in GAIA Expert System(GES) is given. In GAIA Graph evaluater(GGE) section, the established CDFG model of GAIA system and several graph related tasks on which GGE performs are given. The solution to recurring subgraph problem is also proposed.

In Chapter 3, GAIA explorer is executed on a set of applications from literature, and the results are presented. Benchmarks with different attributes are chosen to depict the generality of the system.

Chapter 4 concludes with a summary of the results obtained, with a plan on future research directions.

2. THE DESIGN COMPONENTS OF GAIA

Customization process requires to act on intermediate representation of the application which is generally a graph. Several tasks are carried out on the graph such as extraction of recurring subgraphs, distribution of control nodes' attributes to data nodes and satisfaction of data dependency between the nodes. Expert system is employed to vote on the customization of the application depending on the attributes of the nodes: 1) attributes possessed from the details of the node itself 2) attributes possessed from the place the node resides in the graph. Fuzzy logic on the other hand, is introduced to represent the imprecise terms of customization process for the expert system.

In order to accomplish the above tasks, GAIA Explorer consists of two subcomponents: 1)the fuzzy expert system which acquires the rules of thumb for the customization process; 2) the graph evaluator part which acts on CDFG of the application and evaluates graph related data. In this chapter, the details of subcomponents, namely GAIA Expert System(GES)(section 2.1), and GAIA Graph Evaluater(GGE)(section 2.2) are given.

Main flow could be depicted as follows: Input CDFG data is received by GGE and the data structures to represent the CDFG is created within. Before passing DFG related data to GES, GGE performs several routines in depth-first search (DFS) and breadth-first search (BFS) manner. GGE employs GES both during string matching and during CI/PE selection. In string matching phase, strings created by GGE are passed to the GES system in order to be grouped and counted. GES passes back the string processing and grouping results, so that GGE marks the nodes in pattern in CDFG data structures. Node data are passed from GGE to GES and GES performs the fuzzy expert system task, altering the memberships of the nodes for processor customization. The results are passed back to GGE, and the results and statistics are created by GGE finally. The overall schema of GAIA explorer is given in Figure 2.1. The tasks performed by GES and GGE are mentioned in figure along with the

interaction points in-between the two sub-components.

2.1. GAIA Expert System (GES)

GES is the expert system database of GAIA Explorer. An expert system is a computer software which reacts as a human expert to reach at solutions for the given problems. Being a subsection of artificial intelligence (AI), it employs *facts* to specify the situation under examination and *rules* to act on these facts that correspond to the knowledge extracted in a specific domain. Expert system uses an "inference engine" which organizes the *rules* and *facts* to reach at the results. The inference engine does not have to act on rules and facts sequentially. It employs a specified method to reach at conclusions, like depth-first or breadth-first search, or an approach acting on some *salience factor* dependent upon the *rules*.

As the programming language for the expert system, *FuzzyClips* [28] is used. *FuzzyClips* is an enhancement of *Clips* [29], the expert system development tool created by NASA. *Clips* is developed in C and can be embedded in C programs, motivating the utility of this particular tool set in this work. It employs LISP, a non-sequential rule-based expert system language. *FuzzyClips* development set provides *Clips* expert system tool set with additional fuzzy rules and facts. It is backward compatible with most of the features of *Clips*.

In the literature of custom instruction identification and selection, there has not been any observed approach with an expert system utilized in the process. However, expert systems has been spotted in hardware software codesign area, which can be regarded as the parent branch of CI selection but is in a larger scale [25], [26], [27]. The common point of the works is the choice of large chunks from the system as dictated by the HW/SW codesign process. The partitioning process results in the whole system and control of the system is not guided by instruction set. In this thesis, expert system is employed to perform custom instruction selection phase.

As stated in a recent book about customizable embedded processors [1], the aim

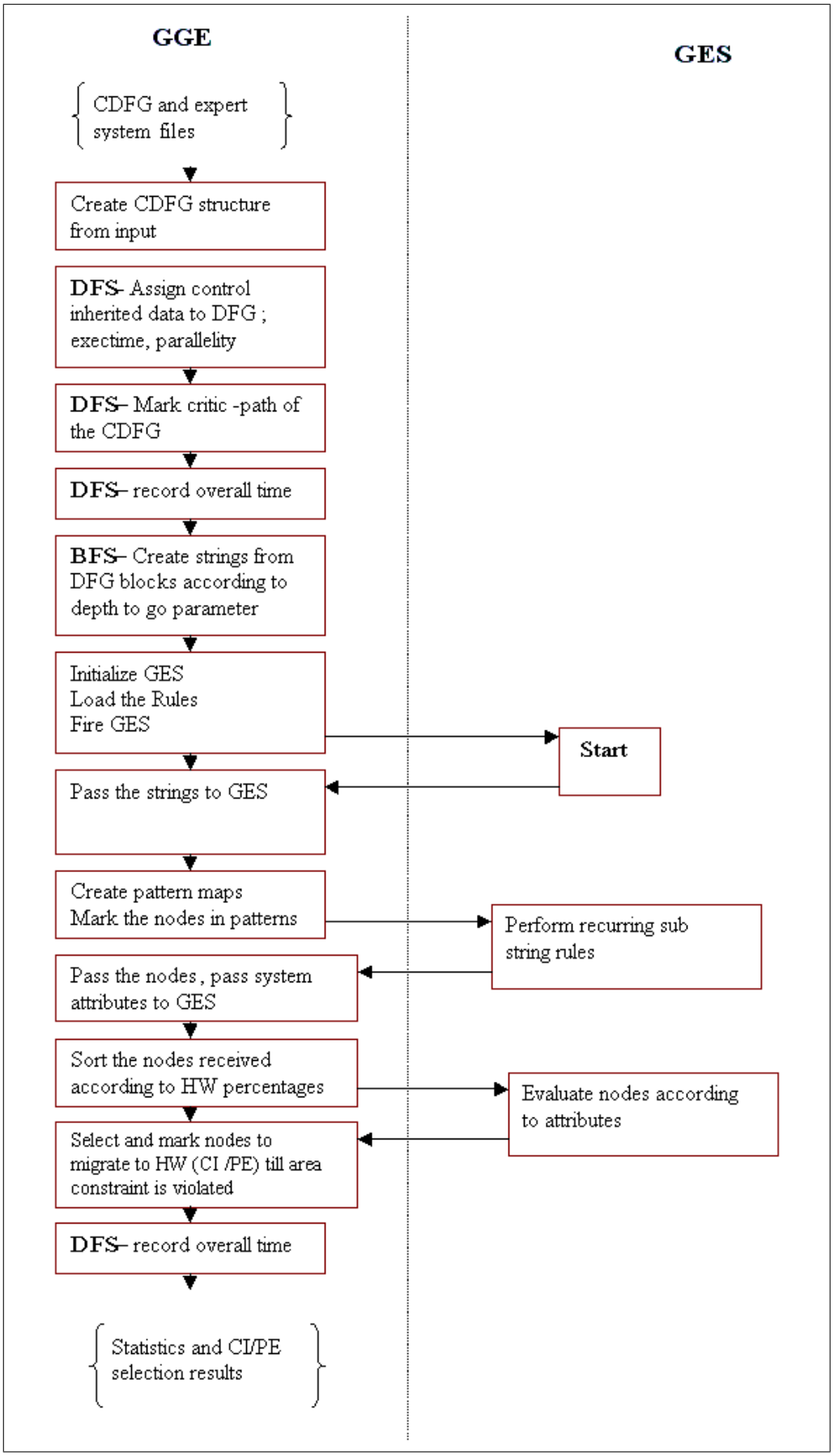

Figure 2.1. GAIA explorer overview schema

of the customization work carried out in literature is to "mimic the choices of an expert". It is observed that the works aimed optimizing an aspect of the CI selection or identification, bypassing the other aspects. This work differs mainly in the idea to introduce a broader range of aspects with hardly optimum features, as an human expert is likely to perform.

Rules and facts of knowledge database are generally acquired to a system by an expert or a knowledge engineer who extracts data from experience of the experts and statistics and transfer them to a database. There are simple databases that provide a GUI to end-user to alter the database. Or it might be the case where the end user has no initiative in the knowledge database and the rules and facts have been coded previously and permanently before. It is also possible that the expert system can be self-learning with an AI method. The rules may be altered by a learning system and be merged into the database.

Fuzzy logic is chosen as a result of the fact that the crisp facts would not be enough to evaluate and vote for the attributes of the nodes/instructions. When fuzzy logic is employed, the attributes of the nodes act together to identify the membership of nodes to architectural components. The attributes are the pre-assigned properties of a node like the execution time of the node, concurrency and being part of a regular pattern. Pre-assigned attributes are initial *facts* of the expert system. When these initial attributes are considered in conjunction with each other, they help the expert system to identify the post attributes, namely the architectural specifications of the nodes.

The architectural specifications of a node are whether it is in HW or SW. If the node is in HW it can either be a CI or be in PE. These attributes are conflicting; for example a node cannot be both in HW and in GPP. This is where fuzzy logic becomes necessary. A node can have membership to any conflicting or cooperating architecture attributes. For example it can be both in HW and in GPP to some extent. However, during the defuzzification process, the most likely attribute is determined with an appropriate technique. An overview chart of expert system is given in Figure 2.2. The

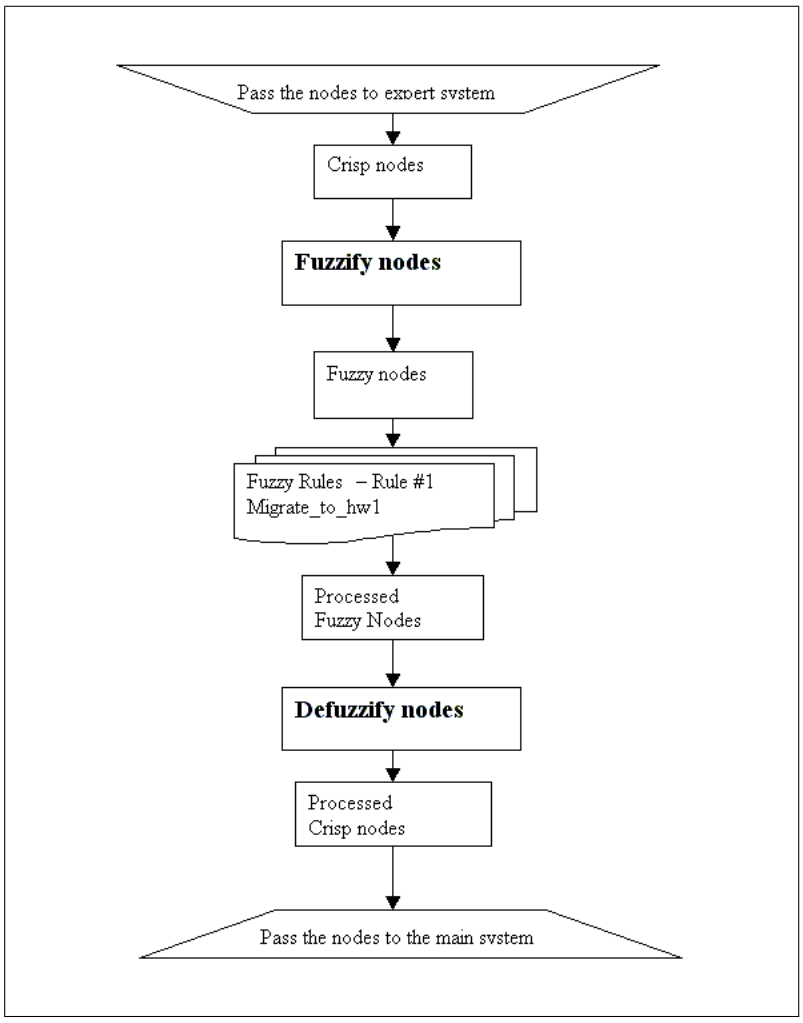

Figure 2.2. Expert system overview schema

nodes are passed to the GES with crisp values. Fuzzification phase is performed to determine the fuzzy values of the nodes. Rules of GES alter the memberships of the fuzzy nodes. Defuzzification phase takes place on processed fuzzy nodes to reach at crisp nodes. Processed nodes are passed back to GGE. The node templates and rule definitions are coded in FuzzyClips language in a ".clp" file; nodes.clp to constitute an input to the system.

For fuzzy attributes of the nodes, membership values are defined with primary terms along with the definition of the range of each primary term. It is also possible to bind fuzzy value membership to a function. Primary terms for memberships of the nodes to the attributes are "low" and "high". For example, if the fuzzy values were created for an age classifier those primary terms would be "young" and "old".

```
(deffunction most-extremely-fcn (?x)
 (** ?x 5)
```

Figure 2.3. An example of a new hedge definition: modify value to the power of 5

With modifiers called "hedges", primary terms are shaped into more precise terms, like "very high" or "slightly high". Hedges are functions that modify the membership values. It is possible to define new hedges by implementing functions and assigning linguistic terms to them. An example of a new hedge definition is given in Figure 2.3. The "most-extremely" hedge has been defined and introduced to the system modifying membership values to the power of five.

2.1.1. Matching of Fuzzy Facts

Rules in *FuzzyClips* consists of two parts; first part being the Left Hand Side(LHS) and the second one Right Hand Side(RHS) of the rule. In the LHS, there exists patterns to match with the facts in the database. If LHS has a match with a fact, the rule is fired and the statements in the RHS are executed. RHS stands for the action rules of fact, may be a function call or a modification to a fact.

Compositional Rule of Inference

LHS of a rule should match with an attribute of the fact in database as determined by the pattern in the LHS. Matching of crisp values is straightforward as it corresponds to equality. For fuzzy variables, equality is not the valid. A fuzzy variable in the LHS part of the rule means it should overlap with a fuzzy fact. For example, the fuzzy facts *pressure low* and *pressure medium* may have a range in common so that they overlap in one range and thus match. The behaviour is called "Compositional rule of Inference". The fuzzy value in the RHS of the rule is determined by the overlapping range of the fuzzy fact in the LHS. The relation can be stated as given in Figure 2.4 .

In Figures 2.5 and 2.6, compositional rule of inference effect on fuzzy value is given. The X *axis* is the per centage of the attribute which is a crisp value. The Y *axis*

stands for the membership values of the nodes, where zero stands for no membership and one for full membership. The sets are depicted with the same X and Y planes for proceeding fuzzy values. The fuzzy value altered is the HW membership which is depicted. The example rule that is altering the HW membership is based on software time (swtime) fuzzy value of the node. The value that is assigned to the RHS of the fact is in Figure 2.5. However, because the RHS of the figure does not fully contribute to the "high" fuzzy definition of swtime attribute, compositional rule of inference determines the LHS. The value assigned to the HW attribute can be seen in Figure 2.6.

The effect of the overlap range differs according to the method used in compositional rule of inference. *FuzzyClips* provides two of them; min-max method and max-prod method [28]. The effect of the two different methods is given in Figure 2.6 for min-max and in Figure 2.7 for max-prod.

P and P'
Observation: X has property P
Relation 1 : X and Y are in relation W1
Conclusion : Y has property Q
Observation: X has property P'
Relation 1 : X and Y s are in relation W1
Conclusion : Y has property Q'

Figure 2.4. Compositional rule of inference

It is not necessary for a fact to fully contribute to a rule. The RHS of the rule is determined by using the method of compositional rule of inference. Compositional rule of inference gives accurate flexibility for voting mechanism of the rules. In this thesis the max-prod compositional rule of inference method is chosen instead of the min-max method because a smoother result on the modified fuzzy value is obtained.

In Figure 2.8 the initial membership of a node, "high-low" is given. "High-low" is a linguistic term that is built up from conjunction of two primary terms: "high" and

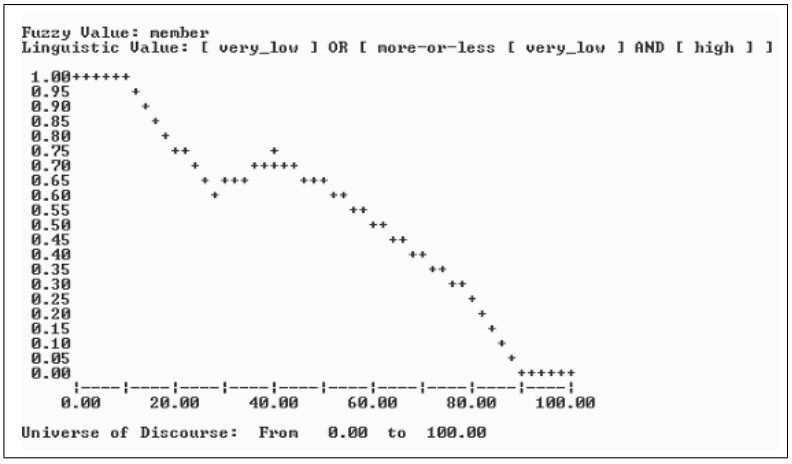

Figure 2.5. Value (Q) determined by the RHS of the rule

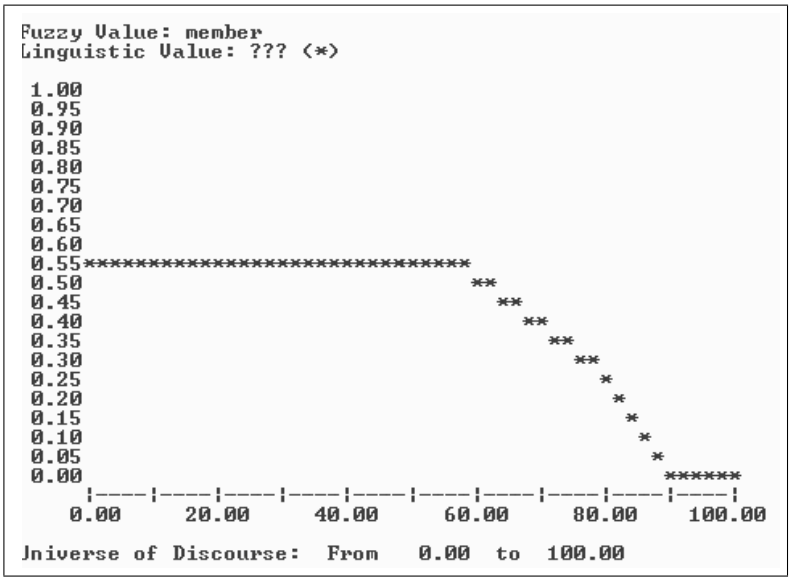

Figure 2.6. Value (Q') determined by the LHS of the rule - min-max inference type

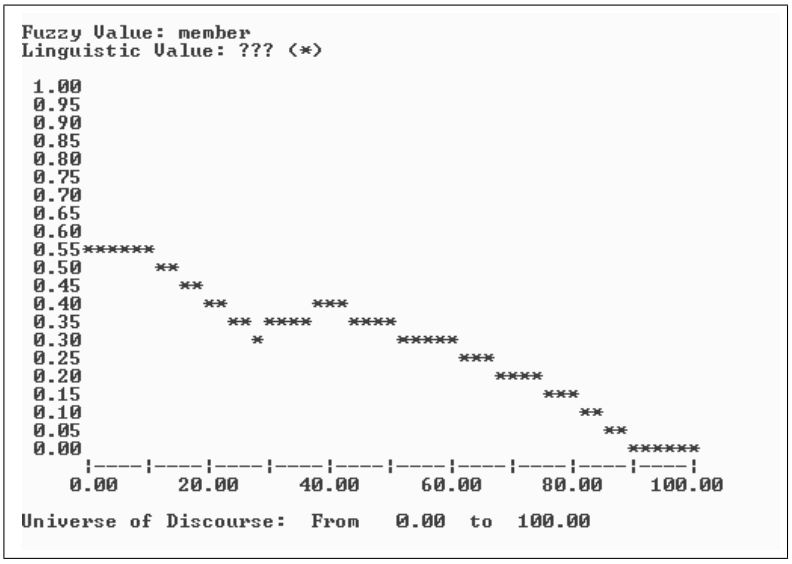

Figure 2.7. Value (Q') determined by the LHS of the rule - max-prod inference type

"low". An inbetween state of this node is given in Figure 2.9, where a HW favoring rule has voted for the membership. String "???" is for fuzzy values that can not be represented with a linguistic term such as "very high".

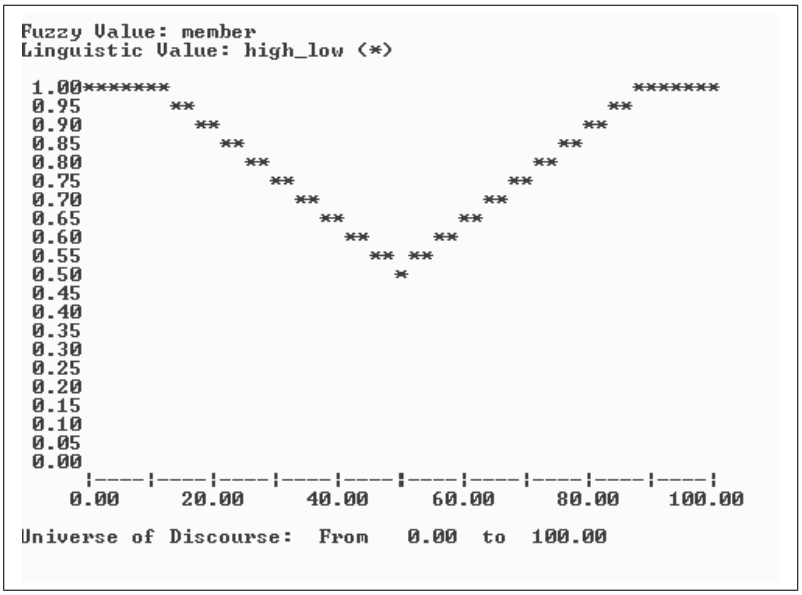

Figure 2.8. Initial HW membership value of the a node; high-low

2.1.2. Defuzzification

After fuzzy rules have fired and altered the HW and PE memberships of the nodes, defuzzification phase takes place. In order to pass the nodes back to the GGE,

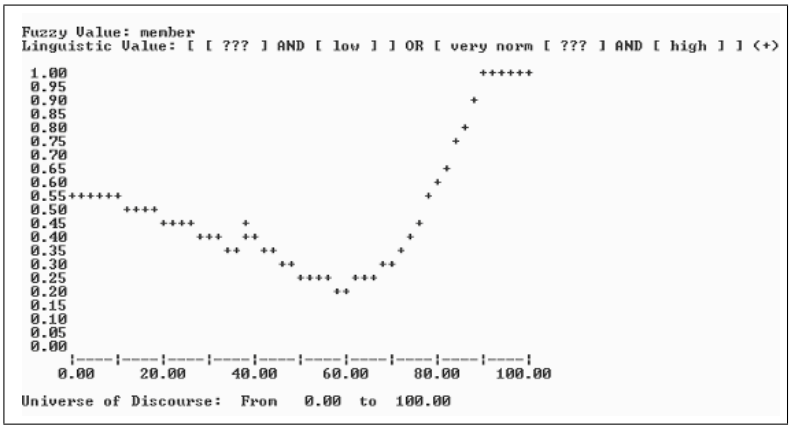

Figure 2.9. Processed node with HW favoring rules:After executions favor HW attribute increase on the same node - max-prod inference type

crisp values corresponding to the membership values have to be calculated. Center of gravity(COG) method has been chosen for defuzzification because mean-of-maxima leads to ambiguity as the membership functions have maximum values at several points. COG method returns the X *axis* value corresponding to the COG of the area under the membership function [28].

2.1.3. Platform and FuzzyClips Compilation

FuzzyClips source code is developed and provided by Orchard [28] for the developers who embed a fuzzy expert system to the system under development. *FuzzyClips* consists of C source code that can be compiled with several operating systems and on several machines. In this work it has first been tried to compile *FuzzyClips* in Microsoft Visual C++ but due to compatibility problems, the platform has been changed to ECLIPSE CDT (C development tool) [30]. For the compiler and linker tool suit, mingw [31] and a GNU (GCC, MAKE, GDB, G++ etc) C /C++ tool suit on Windows platform has been instrumented within ECLIPSE CDT.

In order to tailor the compilation of *FuzzyClips* one has to change the *setup.h* accordingly. The flag for the platform has been chosen as "IBM_MSC" (IBM PC, with Microsoft C 6.0). Also other flags to note in *setup.h* is "RUN_TIME" which is set for compiling the constructs (facts and rules). Compiling the facts and rules

creates a standalone executable, preventing further modification to the rules. When the flag is not set, the rules and templates are interpreted from ascii files. In this thesis "RUN_TIME" flag has not been set so the rules and templates are not stabilized and can be modified. The facts however constructed during execution and is not read from files. Templates and facts required for the system reside in *nodes.clp* and *pattern.clp*. Flags for supporting fuzzy *rules*, *facts* and *templates* in the *setup.h* file has been raised for *FuzzyClips* to be functional.

After *FuzzyClips* source code has been compiled into objects, the system can be integrated to *FuzzyClips* features with required directives. It is utilized basically by including the *FuzzyClips* and *Clips* related headers to the application. Whenever expert system is to be used initialization of the expert system is performed within main application. The facts are placed on agenda as soon as facts are fired. It is possible to create facts and rules, and employ all the other features *Clips* and *FuzzyClips* provides within the application code thorough utility functions.

The design includes both C and C++ langugage components. The design part of graph related functions is in C++ code and the *FuzzyClips* source code is mainly in C. Objects are compiled in their own style and C parts are linked to C++ objects with *C extern* directive later on.

2.1.4. Operator Nodes as Facts

GES evaluates the attributes of nodes in a discrete manner. Group of nodes are handled by the graph related C++ part of the object and required data is propagated to the node attributes.

One of the basic reasons of embedding *FuzzyClips* into graph related C++ code is that the expert system alone is not efficient to assess all the attributes of the nodes. It is prolific in node basis, however some attributes of the nodes require relationship with other nodes. They are inherited from the topology of the application. They can be derived by utilizing graph functions in the graph evaluator of the system. For example,

detecting patterns in the application is a graph function. Based on the pattern data constructed by the graph evaluator GGE, nodes which are found in regular patterns are marked as CIs or PEs by GES. CI and PE selection of nodes is done by the expert system via changing their membership values.

A node template is given in Figure 2.10 which is created for fuzzified nodes. Slots store the fuzzified values of nodes, both pre-assigned and evaluated by GGE or GES. A node stores the values listed in Table 2.1.

```
(deftemplate fuzzified-nodes
   (slot nodenumber)
   (slot HW (type FUZZY-VALUE member))
   (slot SW (type FUZZY-VALUE member))
    …
    …
   (slot PE (type FUZZY-VALUE member))
   (multislot control (default NIL))
   (slot swtime(type FUZZY-VALUE swt_member) )
   (slot hwarea (type FUZZY-VALUE hw_member) )
   (slot execnumber(type FUZZY-VALUE exec_member) )
   (slot parallelity)
   (slot on_critical_path
   (slot special_node)
   (slot in_pattern (type FUZZY-VALUE pat_member) )

  );; end of deftemplate
```

Figure 2.10. Fuzzified node template in expert system

After a fuzzified node has been created from a crisp node, the rules vote for the attributes of the node. The resultant fuzzified node for the example template of Figure 2.10, is shown in Figure 2.11. Note that "???" string is displayed in place of fuzzy values whenever it is not possible to represent the fuzzy value with a linguistic term like "very high". Here, the "high" linguistic term is a pre-defined fuzzy value template that represent the membership set of the "high" term. The ranges for the fuzzy value templates are defined previously in *nodes.clp*. The membership set representation can be seen in the screenshot 2.11 from GES. Membership set with a sequence of numbers is one of the ways to represent fuzzy values. In the screenshot, the first sequence from 10 to 90 represent the membership fuzzy value of HW slot, while the rest follow in order of the slots. It is observed the HW membership value has been altered by rule(s) and the membership set has been recalculated. On the other hand the second set from 10 to 90 has the initial value.

Table 2.1. Slots of the node template

Slot Name	Explanation
HW	Hardware membership fuzzy value determined by GES
SW	GPP membership fuzzy value determined by GES
PE	PE membership fuzzy value determined by GES
control	Storage for control flags of rules
swtime	Fuzzified Software execution time
hwarea	Fuzzified hardware area
execnumber	Fuzzified execution number determined by GGE
parallelity	Concurrency indication determined by GGE
on_critical_path	Indication on being in critical-path determined by GGE
special_node	Indication for special nodes such as IP cores
in_pattern	Fuzzified Pattern count determined by GGE

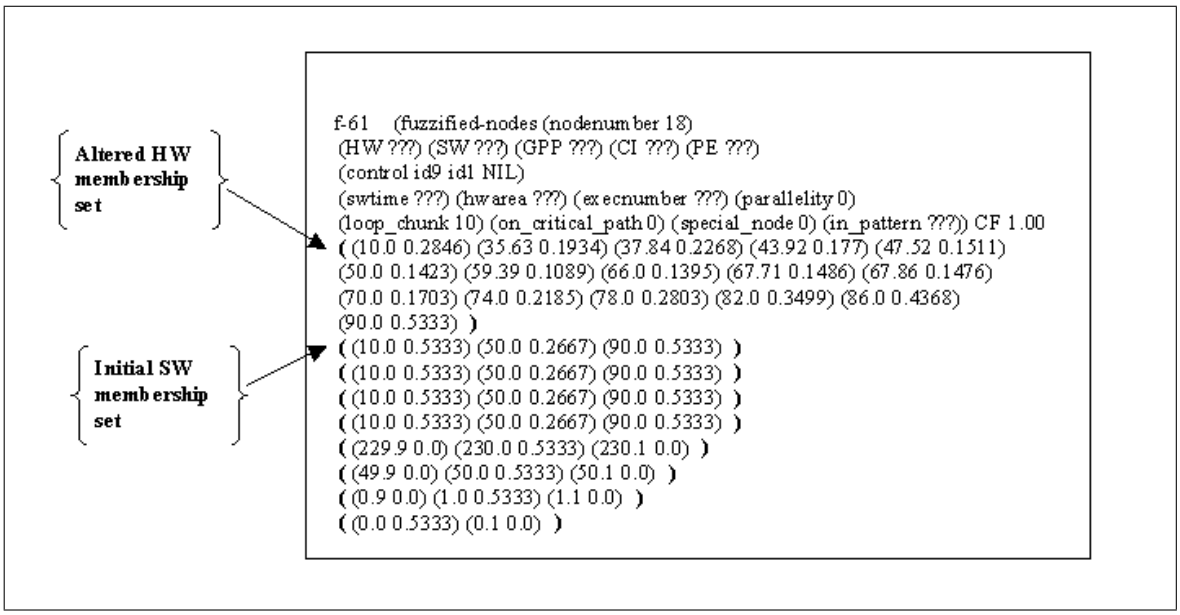

Figure 2.11. A Fuzzified node data example

In Figure 2.12, a rule definition that has been instantiated to change the attributes of the nodes is given. The left side of the "=>" operator stands for THE LHS, while the right side for RHS of the rule. The variables are declared with "?" operator, where the assignment operator is "< −". The usage of the FuzzyClips grammar is given in the tutorial [28]. Clips grammar is given in [32].

```
(defrule migrate-to-hw7
 ?f <-(fuzzified-nodes (nodenumber ?n)
                       (on_critical_path 1)
                       (HW ?HW)
                       (control $?con)

                       )
                       (reduce wcet)

( not (fuzzified-nodes (nodenumber ?n) (control $? id7 $? )) )

  =>

(bind ?highcomp (fuzzy-intersection ?HW (create-fuzzy-value member high) ))

(bind ?highcomp (fuzzy-modify ?highcomp norm ))
(bind ?highcomp (fuzzy-modify ?highcomp very ))
(bind ?lowcomp (fuzzy-intersection ?HW (create-fuzzy-value member low) ))
( bind ?newhw (fuzzy-union ?lowcomp ?highcomp))
(plot-fuzzy-value t + nil nil ?newhw)
( bind ?newf (modify  ?f (HW ?newhw)))
(modify ?newf ( control (insert$ ?con 1 id7) ))
)
```

Figure 2.12. FuzzyClips rule example

2.1.5. Control System for GES

A control system to rule database is mandatory. Modifying nodes in *FuzzyClips* results in modified facts to be evaluated with the same rules again, causing infinite loops. There is no sequential execution in *Clips*. Matched facts and rules are placed in agenda to be executed. Prevention of infinite loops is handled with facts that serve as control flags. In GES, every rule should act upon every fact only once. All the rules matching previously, match again after rule modification because of LHS has patterns of unchanged attributes.

In order to prevent continuous firing of rules a control mechanism has been implemented. This is achieved by introducing control slots to nodes, where every rule evaluating a fact inserts its id to the control slot. Evaluating the control slot prevents the rule to match a node and change its membership a second time. LHS part of the rules include their own flag ID. If the ID exists on some node, the RHS part of the rule, is never operated on that node.

2.1.6. Salience for Expert State Machine Control

Separate stages of the fuzzy system like defuzzification, fuzzification and fuzzy rule attribute evaluation, are manipulated by the method called *Salience*. *Salience* is the term that represents the priority of a rule. High salienced rules fire first while the low ones fire last. Salience is used just for determination of stages in GES. Membership altering fuzzy rules have the same firing salience. The order of the rules with the same priority are set using a breadth-first method: A term standing for newly activated rules have less priority and rules fire in a queue manner.

2.2. GAIA Graph Evaluator (GGE)

An application can be represented as a CDFG where data dependency and control flow are constructed as a directed acyclic graph (DAG). It is assumed CDFG input is in single assignment(SA) model which is constructed from a high level language. The GAIA system acts upon a CDFG, which is a group of nodes that have a pattern according to the rules explained in section 2.2.1.

The GAIA expert system acts on node basis, but nodes have attributes that they inherit from the topology of the graph or the specifications of the application. These attributes are marked by the graph evaluator component of GAIA, i.e GGE.

The graph evaluator component has been implemented in C++ with a generic graph library, since the graph tasks are computation-intensive. A well known and credited work that performs graph related tasks is the BOOST graph library [33]. By utilizing this library, it is possible to merge customized data structures to templates and use libraries functions and algorithms.

GAIA graph explorer assumes there is an application that has been passed to GAIA in the structure of a CDFG. GGE acts upon the CDFG model established in Section 2.2.1.

2.2.1. The CDFG Model

The input of the GAIA system is a CDFG that represents the control and data dependency of operations. A CDFG consists of data nodes and control nodes such as loop, conditions, branches. Group of data nodes correspond to a DFG. A DFG is a directed acyclic graph that has operators as the nodes and operands as the edges. DFG nodes can be fine-grained as addition, subtraction operations or coarse-grained like functions declared in a high level language. There is no restriction on the granularity of the nodes. The coarse grained nodes enables the user to specify special nodes. The coarse grained nodes correspond to functions that have third party netlist files. They can be intellectual property (IP) cores.

It is assumed that the DFG is arranged in SA model which stands for the arrangement of the graph where variables in the graph are assigned only once. The model prevents cycles in a DFG occurring from data dependency. For example the following code can be considered:

1: X ← a+b
2: X ← c+X

Here, X is assigned several times thorough out the sequence. The resultant DFG has cycles due to multiple assignment of the X variable. Creating the temporal register X1 prevents cycles in DFG as well as retaining the data dependency. The resultant model is SA model. The DFGs before and after regulation are given in Figure 2.13. The sequence can be rewritten as below with the X1 variable:

1: X1 ← a+b
2: X ← c+X1

CDFGs can be derived from different high level languages such as C or a hardware description language like VHSIC Hardware Description Language(VHDL). CDFG is also the intermediate representation of compilers, reproduced by the front-end compilation stage. Although the intermediate representation is a CDFG, there is no standard

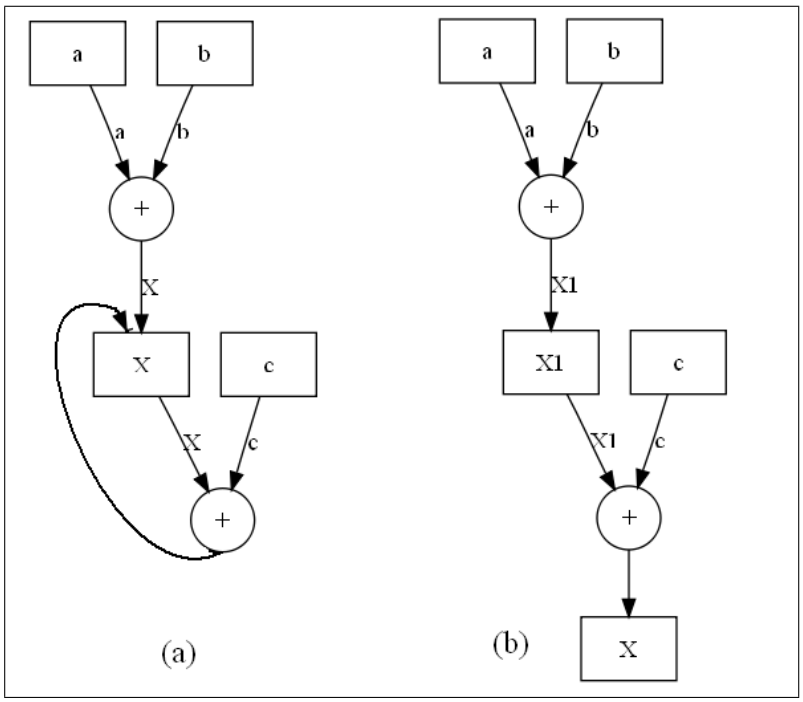

Figure 2.13. SA model regulation a) DFG before regulation b) DFG after regulation in SA model

between the CDFG models [8], [9].

The input of the GAIA system is an abstract CDFG model, which only represents the data needed for CI/PE selection. Evaluating a CDFG representation as the input, the GAIA stands between the front-end compilation and back-end compilation stage. GAIA system should not be integrated after a back-end compilation phase, i.e. after all the nodes have been mapped into assembly language of the target architecture. The necessity arises because of the reason that GAIA system evaluates user-marked special nodes which will be directly mapped to CIs. This is advantageous if the user has a third-party hardware definition of a function; in other words an IP core that will outperform software version, such as square-root function.

CDFG model used in GAIA is defined as follows. CDFG is a directed acyclic graph, G(V,E). The vertex set V of the CDFG consists of two disjoint vertex sets. V_D and V_C stand for vertex sets of data nodes and control nodes respectively. A data node can be a simple arithmetic logic operator or a miscellaneous operator such as a user defined module or an IP. A control node can be one of the following:

- a branch construct like "if/else" or "case/select".
- a loop construct
- a function call

The edge set E of the CDFG is made up of three disjoint sets. E_{DD} is the edge set that represents dependency between data nodes. E_{CD} covers the edges that describe the transactions from control nodes to data nodes. E_{CC} is the edge set that shows the control flow dependency between two subsequent control constructs. Note that there's no edge set from data nodes to control nodes because the CDFG is partitioned into DFGs by using a control node only at the root of each DFG. For a small example, consider the code in Figure 2.14. This code calculates root-mean-square of N values of array *arr*. The corresponding CDFG is shown in Figure 2.15. The operation carried out in line eight is done with a function call, i.e. *average*. The definition of average also appears in the CDFG. The operation of line nine is done with a miscellaneous operator, sqrt.

```
1: void main()
2: {
3: sum ← 0; i←0;
4: while i < N do
5:    sum += arr[i]*arr[i];
6:    i++;
7: end while
8: ave ← average (sum, N);
9: result ← sqrt(ave);
10: }
11: average(sum, N)
12: {
13: return (sum/N);
14: }
```

Figure 2.14. Root-mean-square code

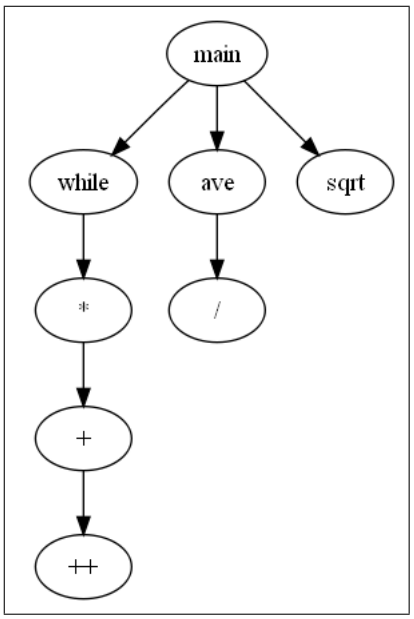

Figure 2.15. Root-mean-square CDFG

2.2.2. The Graph Input

The CDFG graph is the input to the program via several text files consistent with the model. There are three files to form the graph.

- *opdet.txt*: This file holds physical properties of each operand. These are hardware area (hwarea), hardware execution time (hwtime), software execution time (swtime). Operands can be fine grained as a GPP instruction or coarse grained as a function specification. The first column is the key of the operand and constitutes an index for *details.txt*
- *details.txt*: This file holds both DFG and CDFG nodes. It requires operands definitions and has an index entry per node to the *opdets.txt*. There are other graph related attributes per node in addition to the attributes taken from *opdet.txt*. The node can be a control or data node, where '1' flag is used for control and '0' flag for data nodes. It can also attain attributes of being parallel or being a special node depending on whether it is a control or data node. In *details.txt*, the main CDFG node of the application should be on top. This is necessary for the GAIA to operate on nodes properly. The top node can also be a *NOP* (no operation).
- *graphcon.txt*: It represents the connectivity of the nodes. The graph connectivity

is implemented as an adjacency list. The parent node is the starting point in the line, while the preceding nodes are children connected to the parent. The line order of the connection relations is not important. The IDs of the nodes in *graphcon.txt* address the *details.txt* for the attributes of the nodes. The key of *details.txt* is used for indexing operands.

The graph in Figure 2.16 is constructed as an input. The input example files for the Toy DFG is given in Figure 2.17.

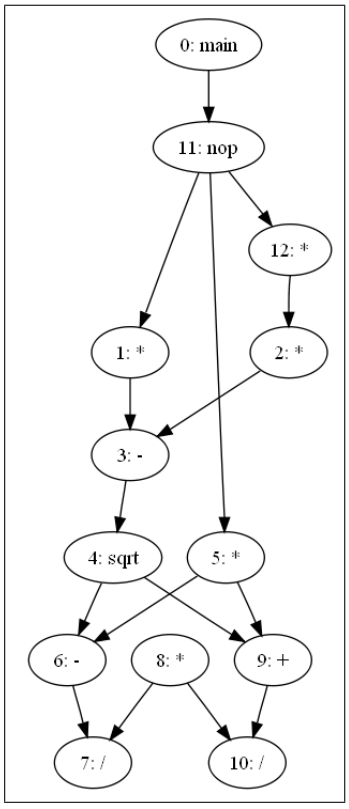

Figure 2.16. The Toy CDFG - DFG part

2.2.3. Boost Graph Library

Boost Graph Library(BGL) is a header-only library that provides data structures related with graphs. Utilities to construct and modify graphs, several graph specific tasks as well as general algorithms used in graph theory are provided by BGL. Boost graph library is designed in the concept of generic programming. A familiarity to standard template programming(STL) is necessary for using the library. With the introduction of templates, flexibilty is targeted. The algorithms of BGL can work on

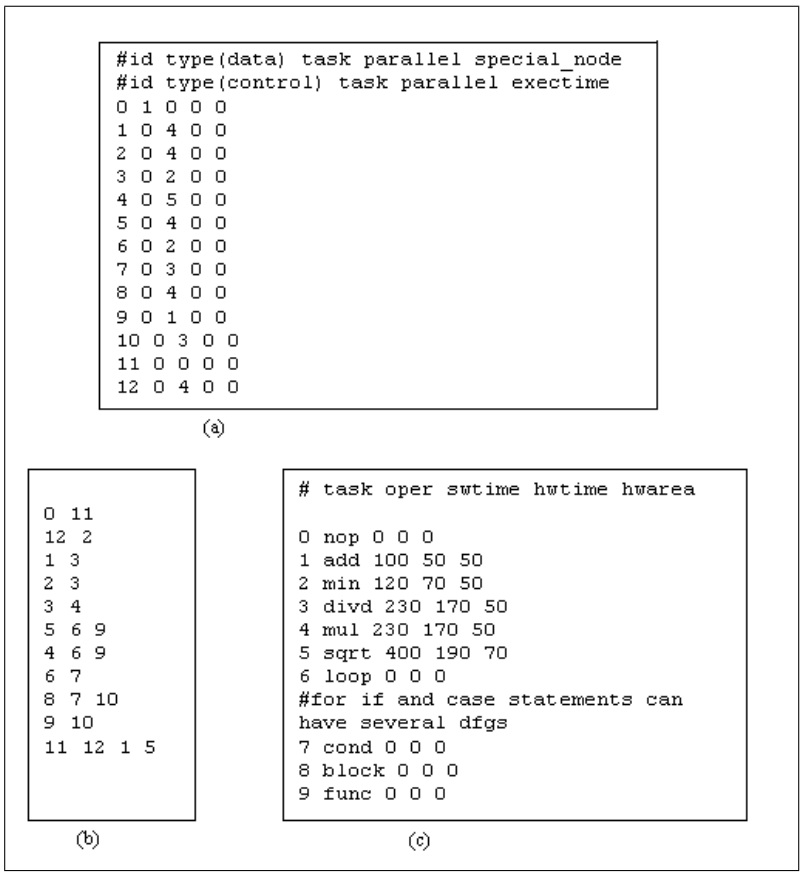

Figure 2.17. The Graph input file examples a) details.txt b)graphcon.txt c) opdets.txt

every kind of data structure. In the thesis boost graph library version boost_1_34_0 is used [34].

The data structure used for the CDFG implementation is an adjacency list instead of adjacency matrix since the graph has a highly sparse structure. In order to enable the movement from child nodes to parents or vice versa, the edges are chosen to be bidirectional. Note that this does not mean undirected or back edges are allowed. Back edges are connections from child to the parent. The resultant graph is acyclic. Introducing bidirectional edges requires additional memory for the GAIA system.

Data structure of vertices has been utilized to hold the data required for nodes of the CDFG. The algorithms are modified to operate on the data included in the vertices, such as BFS and DFS.

2.2.4. Finding Recurring SubGraphs in Graphs

Problem of finding isomorphic subgraphs and a recurring isomorphic subgraphs has many uses in chemistry, biology and data mining fields. It is a highly worked research area. There are several papers on optimization and various algorithms can be classified by the keywords like subgraph isomorphism, recurring subgraphs or motif detection. Most of the work is focused on the topology of the graph, namely the connection between the vertices. The algorithms generally use adjacency matrix of the graph, trying to capture the connection patterns in the graph. Connection patterns usually change. However, in CDFGs, the connection patterns do not change drastically. For the recurring subgraph problem of CDFG, the connections of the vertices are not varied and the graph is highly sparse. The in-edges of a node is generally around two. The in-edges of a node correspond to the input operands, while the node stands for the operator.

The previous work on recurring subgraph problem that has been spotted in HW/SW codesign and CI selection area is as follows. A recurrent subgraph approach has been used in custom instruction identification in a previous work. The approach

acknowledges to find a small patterns [1]. In the approach taken in this thesis, it is possible to identify more than two instructions as well as shift-add and multiply-accumulate chains. It is a brute force algorithm that linearizes the graph for every node and converts the recurring subgraph problem to a recurring sub-string problem, which is less expensive to compute. In this thesis a method has been proposed to solve recurring substring problem. The method is based on Ratcliff/Obershelp algorithm [35] which extracts longest common subsequence between two substrings. The worst case computation time of the algorithm is pseudo-polynomial. Note that when recurring subgraphs are chosen, convexity constraint is considered and introduced into the problem both as a heuristic and as a mandatory constraint for CI patterns.

2.2.4.1. Linearization of the Graph . The approach to subgraph isomorphism detection is implemented with linearization of the graph patterns. It works in a brute-force way, with the addition of a heuristic. The heuristic is that whenever the nodes of the graph violate the convexity constraint, the algorithm stops string formation on that particular node, i.e. newer nodes are not included to the string. Algorithm advances to the next node with a depth-first search performed on the nodes of the DFG. Strings are constructed from every node in the graph. String construction is performed advancing from sink node and including parent nodes upwards. The combination of parents of the node in string formation is also considered since the order of the parent nodes does not make a difference for GAIAs recurring subgraph problem.

One noticeable point of the approach is the spanning of basic blocks for recurrent subgraphs. All the basic blocks are evaluated concurrently in a pool and the recurrency factor is determined on the whole application basis.

The heuristics that reduce the search space are as follows.

- Convexity violation: The parents of the nodes are evaluated to one by one. If the included node violates convexity then route is terminated and string is constructed for only up to that point.

- Control node elimination: Strings are not formed with control nodes. Once a control node is reached, the string generation is stopped. The string generator is working on DFGs only. Note that control nodes and data nodes exist in CDFG. According to the graph model, control nodes are reached from a control node which is the root, aka main.
- "Depth_to_go" parameter: The details of depth_to_go parameter are given in section 2.2.4.2.

The constructed strings after linearization of the graph are transferred to the expert system in order to detect the similar patterns. Similar patterns are asserted as 'pattern facts'. The matched patterns are grouped, counted and marked so that the highly recurring patterns have a higher weight in CI selection process in GES system.

2.2.4.2. Depth of Linearization . Linearization is performed on all nodes, starting from the nodes that are reached in a depth first manner. The task is performed from the starting node up above the ancestors. It is possible to continue string formation until the entrance node of DFG is reached. In this way bigger patterns can be extracted. However this is computationally a heavy process. Therefore a *depth-to-go* parameter is introduced. *Depth-to-go* limits the level that the algorithm can go above, up to the "main". Considering the case *depth-to-go* is two, the resultant level that can be advanced from the "a3" is shown in Figure 2.18. The possible clusters from which the string are formed for node a3 is given in Figure2.19. Corresponding strings for clusters is given in Table 2.2. Aliases of the nodes given in the example are fictive. In the GAIA system, the operators are assigned a unique character and the characters of the operators form the string patterns. For example, considering a3 and a2 are addition operators, an 'A' character replaces "a3" and "a2". The character "!" is employed to represent level changes in the DFG. There are no isomorphic patterns in the example, just possible node linearization for node "a3" is given.

2.2.4.3. Convexity. Convexity of a subgraph is mandatory for Custom instruction selection. Otherwise the operators in the custom instruction would have to produce

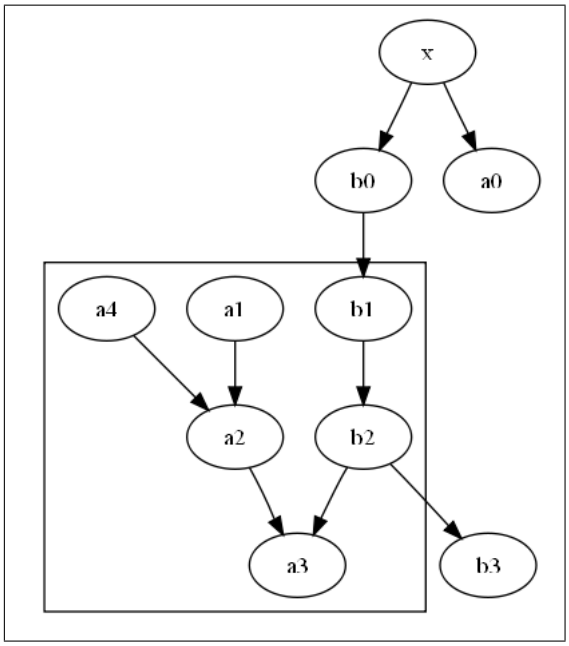

Figure 2.18. Cluster of DFG for a3 node Depth-to-Go = 2

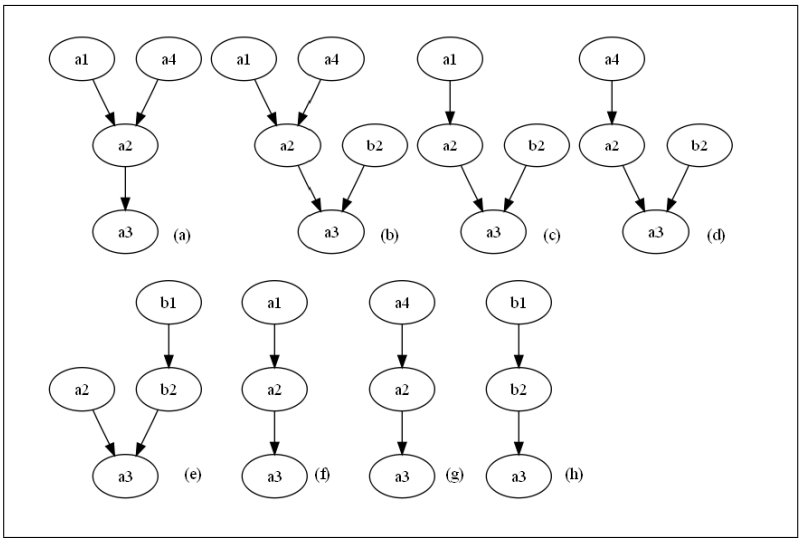

Figure 2.19. Subclusters for a3 node Depth-to-Go = 2

Table 2.2. Strings formed for a3 node

Cluster	Strings formed
(a)	!a3a2!a2a4a1! and !a3a2!a2a1a4
(b)	!a3b2a2!a2a4a1 and !a3b2a2!a2a1a4
(c)	!a3b2a2!a2a1
(d)	!a3b2a2!a2a4
(e)	!a3a2b2!b2b1
(f)	!a3a2!a2a1
(g)	!a3a2!a2a4
(h)	!a3b2!b2b1

data and wait for an external input at some point in the execution. A halt event during the operation of the instruction interrupts atomic execution of the CI. Therefore, convexity constraint should be met, in order for a CI to execute atomically.

Convexity of the subgraph is traced during pattern string generation phase. Whenever strings are generated from the graph, parent vertex is added to the subgraph only if convexity is not violated. If the addition of the parent vertex violates convexity, string formation on that part of the tree is not resumed and aborted.

At the parent addition process, the children nodes of the candidate vertex are checked. The adjacent nodes are children of the parent vertex. For every child that is not already in the subgraph (i.e. CI set), a BFS is performed. The algorithm concludes that the convexity is violated if a node in subgraph, i.e. CI set is visited during the BFS search. The subgraph must be convex prior to the addition. After addition to the subgraph, if the parent vertex under examination results in a path that ends in the subgraph, but has vertices outside of the subgraph, then convexity is violated. The formulation of the convexity algorithm is given in Figure 2.20

FUNC Convexity Checker
INPUT: G_D (V_D,E_{DD}), G_s (V_s, E_s) and $v \epsilon V_s$
such that $G_s \subseteq G_D$, V_s'= $V_D - V_s$ and G_s is convex
FOR every ($w \epsilon V_s$' and u, $z \epsilon V_s$ such that $e_{vz} \epsilon E_{DD}$ and $e_{wu} \epsilon E_{DD}$)
IF there exists a path between v and w, THEN V_s is not convex.

Figure 2.20. Convexity algorithm

2.2.4.4. The Flow of Subgraph Pattern Matching. First step is the generation of the patterns from the BBs of the CDFG in the GGE part of the system. GGE part forms the strings regarding the convexity constraint, control node elimination and depth-to-go parameter. Once the strings are created, they are passed to the expert system so as to be matched and grouped under patterns. Receiving a pool of string patterns, the expert system evaluates them with GES rules. Note that these rules are not fuzzy and made up of string functions mostly.

The string processing function is coded in C and added to the source code of *FuzzyClips* so that it can be used within *FuzzyClips* rules. The function evaluates the input strings two by two and returns all the sub-strings common in both strings. This enables the linearized portions of the graph to be compared one by one. As a result, all common substrings are created. The substrings are then collapsed and grouped with their counts by GES. The resultant group of substrings, along with their counts, is passed to the GGE. A basic flow chart is given in Figure 2.21

2.2.4.5. SubString Matching Algorithm. The algorithm, which finds all matching strings between two strings is developed by modifying Ratcliff/Obershelp pattern-matching algorithm. Ratcliff/Obershelp algorithm finds the longest subsequence between two one-dimensional strings. For two input strings of length n and m, the algorithm creates an (n+1)x(m+1) matrix, as in Ratcliff/Obershelp algorithm [35]. The way matrix entry values are calculated is modified, in order to find matching strings, not sequences. The complexity of the algorithm is $O(M * N)$. The modified algorithm is shown in Figure 2.22. The terms $str1$ and $len1$ stand for the first string and the

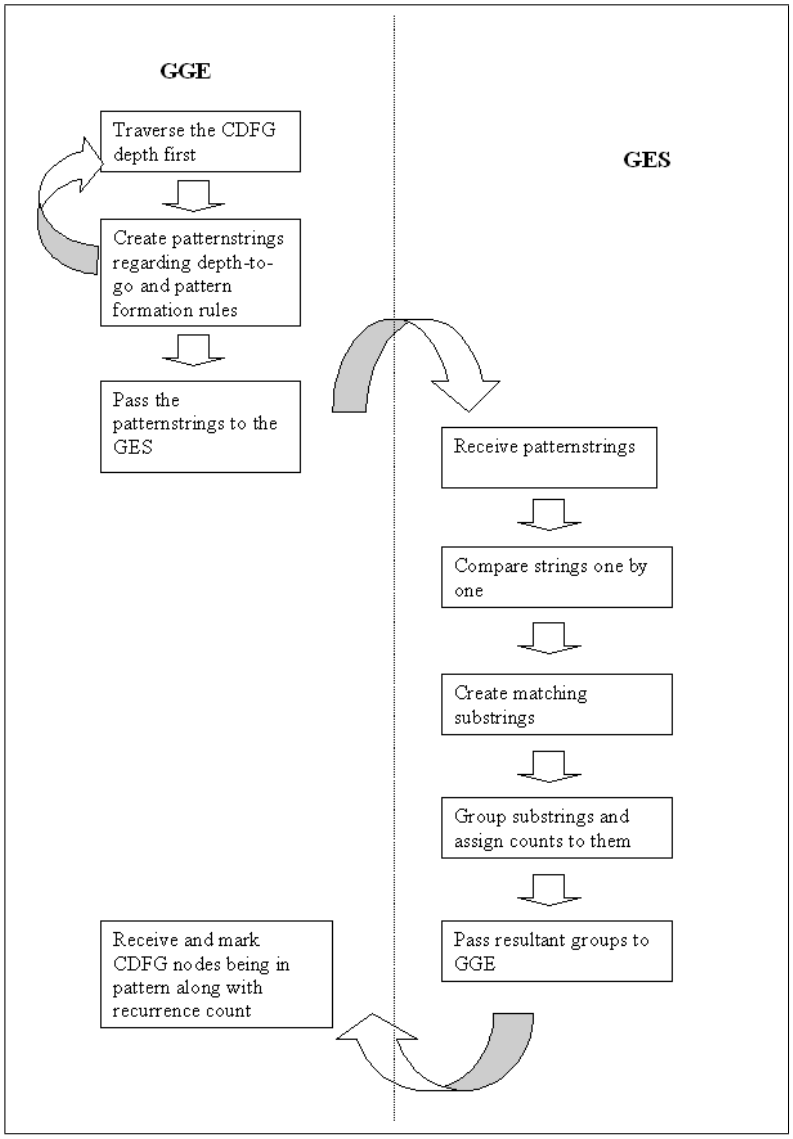

Figure 2.21. String formation system flow

length of the first string. Second string is indexed with two.

```
for(r=1; r<=len2; ++r)
        for(c=1; c<=len1; ++c)
              if( str1[c-1] == str2[r-1])
                    matrix[r][c]=matrix[r-1][c-1]+1;
              else
                    matrix[r][c]=0;
```

Figure 2.22. Modified Ratcliff/Obershelp algorithm

To clarify, if the selected characters do not match, the cell that stands for their combination will be 0. Otherwise, it is calculated by increasing the value in the up-left corner by 1. Since up-left corner stands for the pair of both characters' preceding characters, eventually, each cell will reflect "number of characters matching" up to that pair, in the matching string it is involved. After the matrix is built, every value greater than one is followed towards up-left cells, and the strings are derived. A sample matrix for strings "abcdef" and "bcdxdef" is depicted in Figure 2.23

abcdef
bcxdef

		0	a 1	b 2	c 3	d 4	e 5	f 6
	0	0	0	0	0	0	0	0
b	1	0	0	1	0	0	0	0
c	2	0	0	0	2	0	0	0
x	3	0	0	0	0	0	0	0
d	4	0	0	0	0	1	0	0
e	5	0	0	0	0	0	2	0
f	6	0	0	0	0	0	0	3

Figure 2.23. String matching matrix example

The operation sequence on sample matrix in Figure 2.23 is as follows:

- Value: c,
- Up-left: b. String = bc Value 1 reached.
- Value: e,
- Up-left: d. String = de Value 1 reached.
- Value: f,
- Up-left: e, String = ef Value 1 not reached, go on.
- Up-left: d, String = def Value 1 reached.

- Matching strings found: bc, de, ef, def

2.2.5. Parallelity of the Application

There may be parallelism in an application in the structure of data parallelism or instruction parallelism. It is well known the most profitable one is the data level parallelism to exploit. This is done in SIMD architecture, where same operation executes on data that is separated into chunks. Concurrently executing identical PEs act on different chunks of data. This is possible in loops which are not data dependent or vector operations that act on a data vector that can be divided into subvectors.

The other kind of data parallelism is instruction level parallelism. This is a VLIW architecture where the parallel executing PEs are not identical but they operate concurrently on independent data. A possible way to exploit instruction level parallelism is finding disconnected parts of the CDFG and fusing the operations together where operands are not identical but can be executed in parallel. This approach is taken mostly for vector data since the cascaded instructions can be executed several times on portions of vectors. In general applications, application attributes do not benefit from instruction level parallelism in an extensive order. An example in literature is XTENSA [36] that exploits data and instruction level parallelism together.

2.2.5.1. User Indicated Parallelism in Programs. The parallelism of the program is mainly expressed with the help of user in parallel programming languages via compiler directives. Some well known programming languages are MPI and OPENMP. Alternatively, programming languages can profile the parallelism in the program to some extent. Also in XTENSA system, the user has to mark the parallel portions of the application [36].

It is possible to mark parallel regions or indicate parallel loops in OpenMP via pragmas [37]. Loops are generally the main source of parallelism in applications. For the loops to be executed in parallel, there should be no data dependency between each

```
while i from 1 to 100 do
  a(i) = a(i) + b(i)
end while
```

Figure 2.24. Data independent parallel loop

iteration. Considering the example in Figure 2.24, the iteration could be done 1-50 on one thread and iterations 51- 100 on the other if there were two identical PEs.

2.2.5.2. Vector Parallelism. Another data parallelism to consider is vector instructions, where same operand executes on the vector entries. Vector parallelism can be exploited in SIMD style. VLIW style may be chosen where several operators are cascaded for corresponding architectures. In this thesis, it is focused on the systems where PEs are identical.

2.2.5.3. Exploitation of Parallelism in GAIA. For the reason that data parallelism is the most profitable attribute to exploit and most tools expect the user to mark the parallel parts, GAIA explorer assumes input CDFG has its loops marked as parallel or the vector data nodes marked as parallel if there is any parallelism to exploit.

For data nodes, the parallel operation is restricted only to vector data nodes. In this case, data nodes are separated to PEs. There is no specification for vector or scaler data indication in the DFG model due to abstraction. If a data node is marked as parallel the voting mechanism assumes it is parallelizable. Granularity of data node can be as fine as an operand or as coarse as a user defined function. There is no restriction since GAIA does not care about the granularity of the nodes.

If the loop node is marked as parallel, the DFG portion under the loop control node is marked also parallel by the system. It is the DFG part of the loop that is divided into PEs. Section 2.2.5.4 describes the difference of a PE from a CI. Details of the algorithm carrying the parallelity of the loop node down to data nodes is given in section 2.2.7.4

2.2.5.4. The Difference Between a PE and a CI. This lexicon is taken from the soft processor implementation of Sonmez [21]. When partitions of HW is called as a PE, an SIMD style architecture is meant where parallel PEs reside along with the GPP. PEs are indeed invoked by custom instructions. Whenever CI is mentioned, the architecture is SISD style tightly coupled custom instruction that is working along with GPP.

2.2.6. Node Selection

After the voting for the customization is performed in GES, nodes have to be chosen to be in CI, PE or GPP. The resultant nodes that GES passes to the GGE are sorted according to their HW per centages. The selection process is dominated by GES, whereas GGE performs graph-related fusion. Node fusion occurs when a chosen node is in a recurring pattern. The other nodes in the pattern are also marked to be in CI.

The pattern choice is another criteria since a node can be in several patterns. The default approach is highly recurring pattern to be exploited. To give a penalty for small patterns, the pattern to choose is determined with the following formula: $RecurrencyFactor * NodesInPattern$. $RecurrencyFactor$ is the count of the patterns found in the CDFG. The value is calculated by GGE. $NodesInPattern$ is the number operations residing in a pattern.

The results for the method that favors the biggest patterns are also observed. The criteria for the big pattern choice was simply $NodesInPattern$. This method chooses the pattern which has the most operations in it.

2.2.7. Graph Traversal Algorithms

GGE part of the system performs several tasks on the overall CDFG of the input application. The most notable is the linearization of the graph with string formation task which has been explained in section 2.2.4.1. The other smaller tasks performed in graphs are mentioned in following sections. The input to the graph algorithms is the

```
FUNC BFS CriticPathTraveler
for all nodes in CDFG do
  if CDFG to DFG entrance then
    COMPUTE overall time consumed for current DFG BFS
  end if
end for
GET the DFG subgraph with highest value for DFG time consumption
CALL BFS CriticPathMarker with critic DFG
ENDFUNC

FUNC BFS CriticPathMarker
for all nodes in the input DFG do
  SET node attribute "criticpath"
end for
ENDFUNC
```

Figure 2.25. CriticPath handlers pseudocode

CDFG of the graph, while the output is the processed CDFG unless stated otherwise.

Graph traversals have been implemented with the help of Boost Graph Library visitor concept [33]

2.2.7.1. Critic Path Traveler and Marker. Critic Path traveler is for identifying the DFG that has the most time consuming task in the graph. All the CDFG is traversed to compute the time span of the DFG parts of the graph. After the Critic DFG part is identified with Critic Path Traveler, the nodes in the DFG are marked in "critic-path" by Critic Path Marker. The pseudocode of the algorithm is given in 2.25

2.2.7.2. CDFG String Travelers. :

The functionality of string formation and details about the algorithm have been

```
FUNC DFS CDFGForStringTraveler
for all nodes in CDFG do
  if CDFG to DFG entrance then
    CALL BFS StringConstructorTraveler for the DFG
  end if
end for
ENDFUNC

FUNC BFS StringConstructorTraveler
for all nodes in input DFG do
  CALL CreateString (recursive)
end for
ENDFUNC
```

Figure 2.26. CDFG string handlers pseudocode

identified in section 2.2.4. The pseudocode of the implementation is given in Figure 2.26.

2.2.7.3. Convexity Checker. Convexity checker is called during string formation task, whenever the string formation advances up to parent nodes. The convexity algorithm usage and phase is given in section 2.2.4.3. The implemented pseudocode is depicted in Figure 2.27. The function *ConvexityChecker* is given in 2.20.

2.2.7.4. Loop Inherit Traveler. Loop Inherit Traveler is for propagating some of the attributes of control nodes to the DFG nodes that are the children of the control node under evaluation. For example if a "loop" node is parallel, DFG nodes under the control loop node possess parallelity. The pseudocode of traveler that propagates attributes of control nodes to data nodes is given in Figure 2.28

2.2.7.5. Total Time Traveler. Total time traveler is for statistical data which is the overall execution time the application CDFG consumes. The traveler is executed both

```
FUNC IsStillConvex
SET S as the current Custom Instruction
SET V as the set of adjacent nodes of parent to be added
for all v ϵ V do
  CALL BFS ConvexityChecker for v with S
end for
if BFS ends in a node that is in S then
  RETURN convexity is violated
end if
ENDFUNC
```

Figure 2.27. Convexity pseudocode

```
FUNC DFS LoopInheritTraveler
CurrExectime ⇐ 0
for all nodes in CDFG do
  if advancing from a loop node to a DFG node then
    parallelity ⇐ loop node's parallelity
    currExecTime ⇐ LoopsExecTime * currExecTime
  else
    nodesExecTime ⇐ currExecTime
    nodesParallelity ⇐ parallelity
  end if
  if GOING OUT a pre-visited control loop node then
    ROLLBACK currExecTime and parallelity value
  end if
end for
ENDFUNC
```

Figure 2.28. LoopInheritTraveler pseudocode

```
FUNC TotalTimeTraveler
overallTime ⇐ 0
for all nodes in CDFG do
  if the node is data node then
    if in HW then
      if in CI then
        overallTime+ = (hwTime ∗ execTime)
      end if
      if in PE then
        overallTime+ = (hwTime ∗ execTime)/PE
      end if
    end if
    if in SW then
      overallTime+ = (swTime ∗ execTime)
    end if
  end if
end for
ENDFUNC
```

Figure 2.29. TotalTimeTraveler pseudocode

before and after the CI/PE selection phase. The improvement achieved on the total time can be recorded and compared via Total time traveler. The results can be observed in *statistics.txt*. Note that the statistics are dependent on the user input of "*opdets.txt*" file which includes the timing values entered by the user for hardware execution time and software execution time.

The total time traveler basically performs the functionality as in the pseudocode depicted in Figure 2.29.

3. EXPERIMENTS AND RESULTS

Experiments have been carried out on a cryptographic algorithm Rijndael and an algorithm for motion estimation from MPEG7 specification. Rijndael gives hint on the recurring sub-graph behavior of the GAIA Explorer. Details of the run instances and the results are given in section 3.1. Motion Intensity works on matrices and performs operation on matrix elements, which can exploit parallelism. The details on motion intensity calculation example is given in section 3.2. The run instances are performed on Windows XP Professional OS, Intel P2 1400MHZ processor, 320MB RAM .

3.1. Example: Rijndael Encyption

Rijndael is a cryptographic language that is the new generation of AES, chosen by NIST. It is a block cipher algorithm that can support several key sizes of 128,192 and 256 bits. The round count differs according to the key size. The round count used in the example is 12. The CDFG plot of the Encryption phase of Rijndael is given in Figure 3.1. *Band* label stands for the operator "bitwise and". *Encrypt*, *decrypt* and *getu32* are functions in Rijndael standard. Rijndael CDFG has recurring patterns due to rounds in encryption. The example is chosen with *full_unroll* mode, which is the extensive mode of the standard and increases patterns. The related CDFG files are constructed from C source code.

The number of operator and control nodes in encryption stage is 170. It took GAIA explorer 30 mins to evaluate the CDFG input and create the results.

Pattern selection method has been chosen both as *biggerpatterns* and *highrecurring* favoring. The details of the pattern selection methods is given in Section 2.2.6. The found patterns with their assigned characters are given in Table 3.1. 'M', 'L', 'K' characters stand for ' ^ ', 'band' and '<<' operators in the Figure 3.1, in order. The assigned value for *depth_to_go_parameter* is equal to two. The chosen nodes with big block favoring method, *biggerpatterns* is sketched in Figure 3.2. The

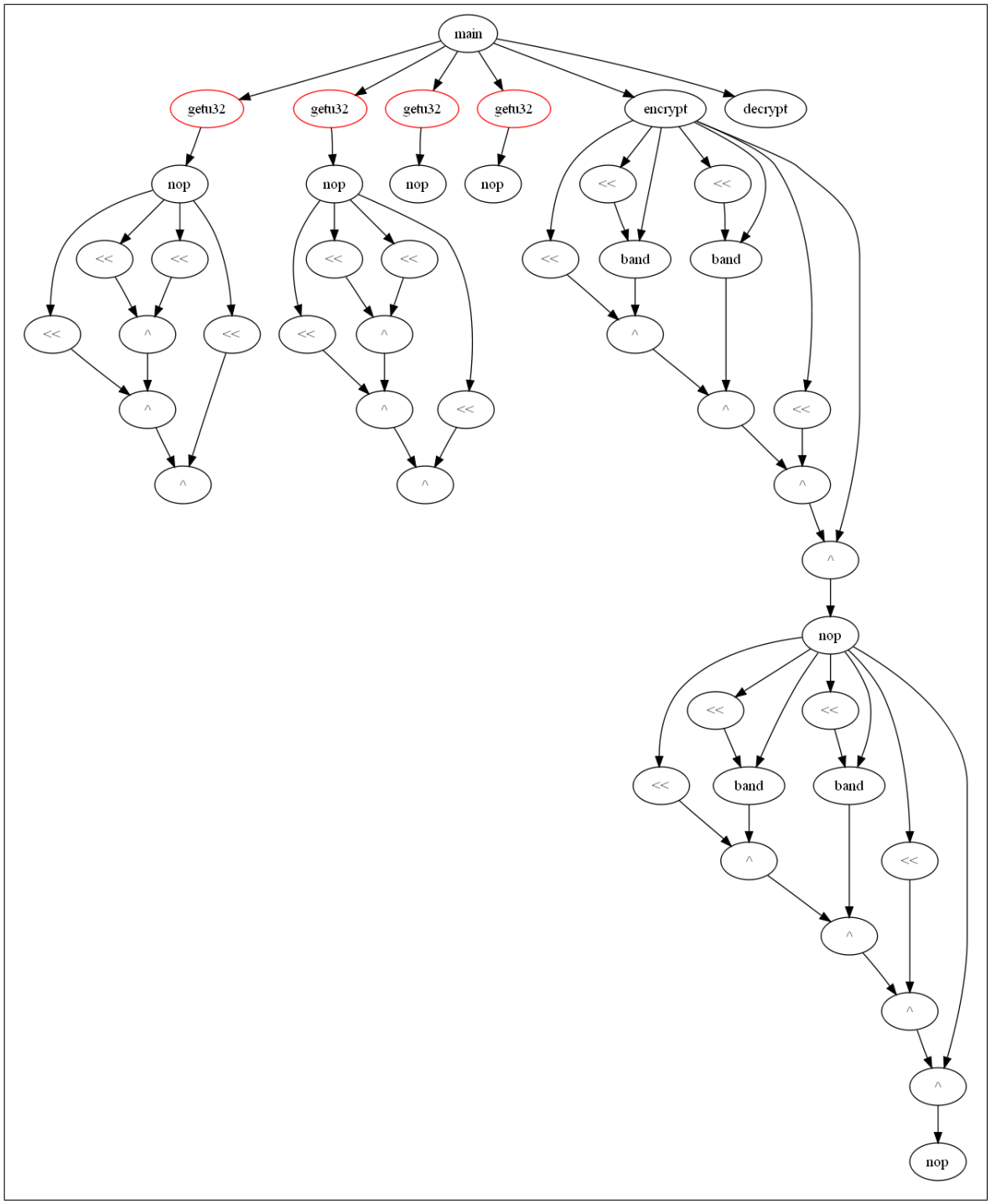

Figure 3.1. Rijndael encryption CDFG

results of *highrecurring* favoring method is given in Figure 3.3. The chosen pattern with *highrecurring* method also exists in *getu32* function DFGs, which is a function found in Rijndael.

Table 3.1. Recurring subPattern results for Rijndael

method	resultant time	pattern
biggerpatterns	25200 *timeunits*	!MLM!MLK
highrecurring	26250 *timeunits*	!MM!MM

High recurrence method, which is basically $recurrencefactor * patternsize$, results in a longer execution time than big block favoring method. However, the results are highly dependent on application under test. Assuming a case, there are two instances of the bigger pattern while there are 100 instances of the small one, it might be favorable to select the high recurrent pattern as the CI.

GAIA Explorer is experimented on Rijndael encryption CDFG for different area contraints. The time consumed for the CDFG when residing in GPP was 28200 time units. The results for different area constraints is given in table 3.2. The pattern choice is based on high recurrency factor of the patterns.

Table 3.2. Experiment results for Rijndael encryption

	Area constraint chosen	resultant execution time
Run Instance 1	1000 *areaunits*	26250 *timeunits*
Run Instance 2	3000 *areaunits*	21150 *timeunits*

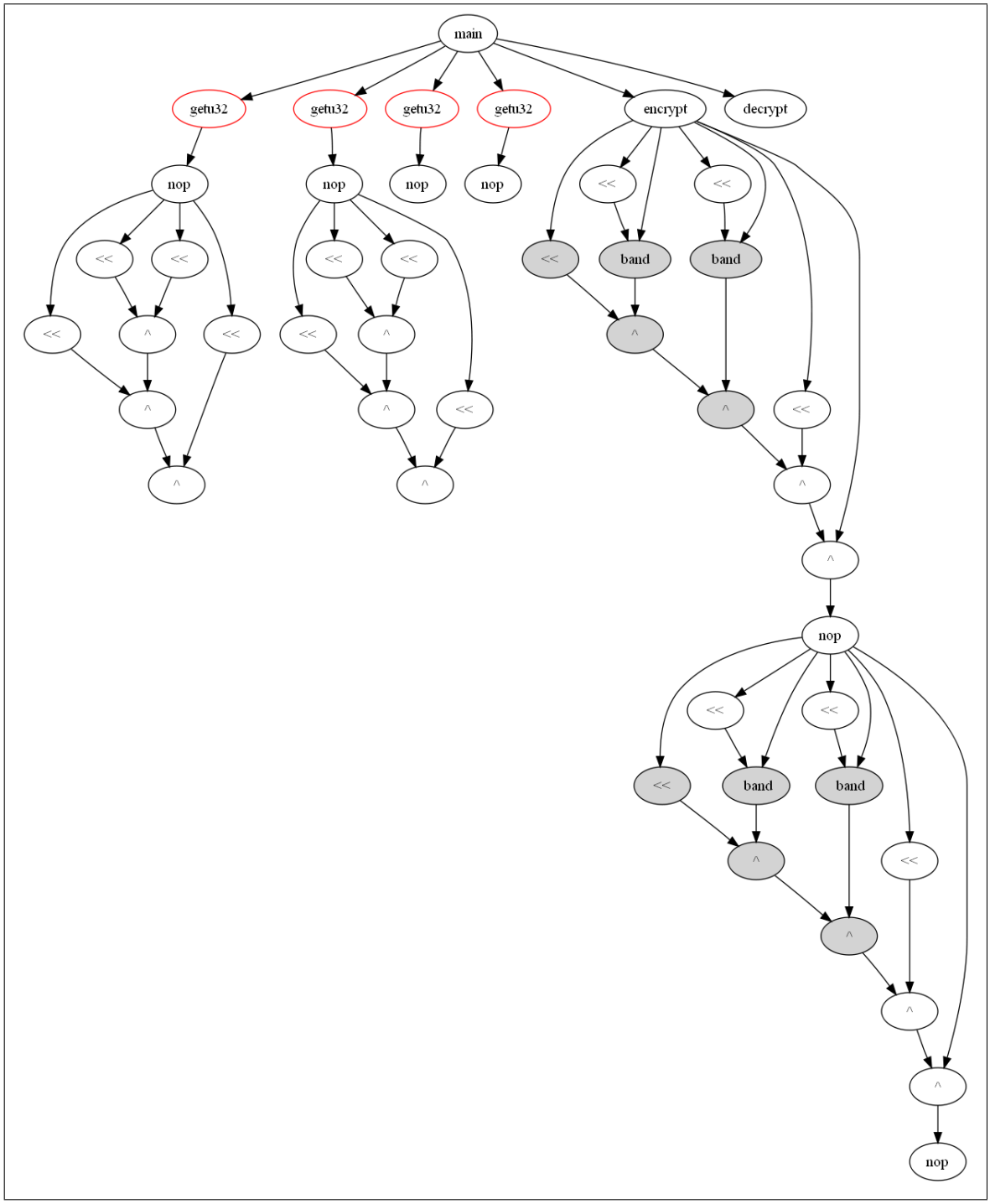

Figure 3.2. Rijndael - Big pattern favoring method; depth_to_go= 2

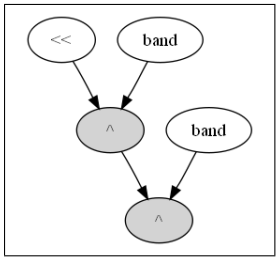

Figure 3.3. Rijndael - High Recurrence factor method; depth_to_go= 2

3.2. Example: Motion Intensity Calculation

The MPEG-7 standard is for providing a framework for handling multimedia content. Color, texture, shape and motion can be represented with a set of descriptors and can be employed in several multimedia tasks. The motion intensity calculation is used for motion estimation in spatial video sequences. GAIA Explorer system is run on motion intensity calculation function which is a part of the MPEG7 motion estimation algorithm. The key point of the motion intensity calculation function is, the function works on matrices to perform the required task. The function can merit from parallelization and as well as from IP cores, for functions like square root. There is no recurrency to exploit in the application

The CDFG representation has been constructed consistent to the established model in section 2.2.1. The CDFG model depiction is given in Figure 3.4. The application consists of 40 control and operation nodes. The matrix size is taken as $12 * 12$. The application takes 40652 time units to execute when all in GPP. The experiment inputs and results after several runs are given in tables 3.3 and 3.4 in order. The GAIA Explorer run time is 5 secs.

Table 3.3. Experiment inputs for motion intensity

	PE count	Area	area critic	special function
Run Instance 1	4	1000 *areaunits*	N	N
Run Instance 2	2	1000 *areaunits*	N	N
Run Instance 3	2	300 *areaunits*	Y	Y (sqrt)

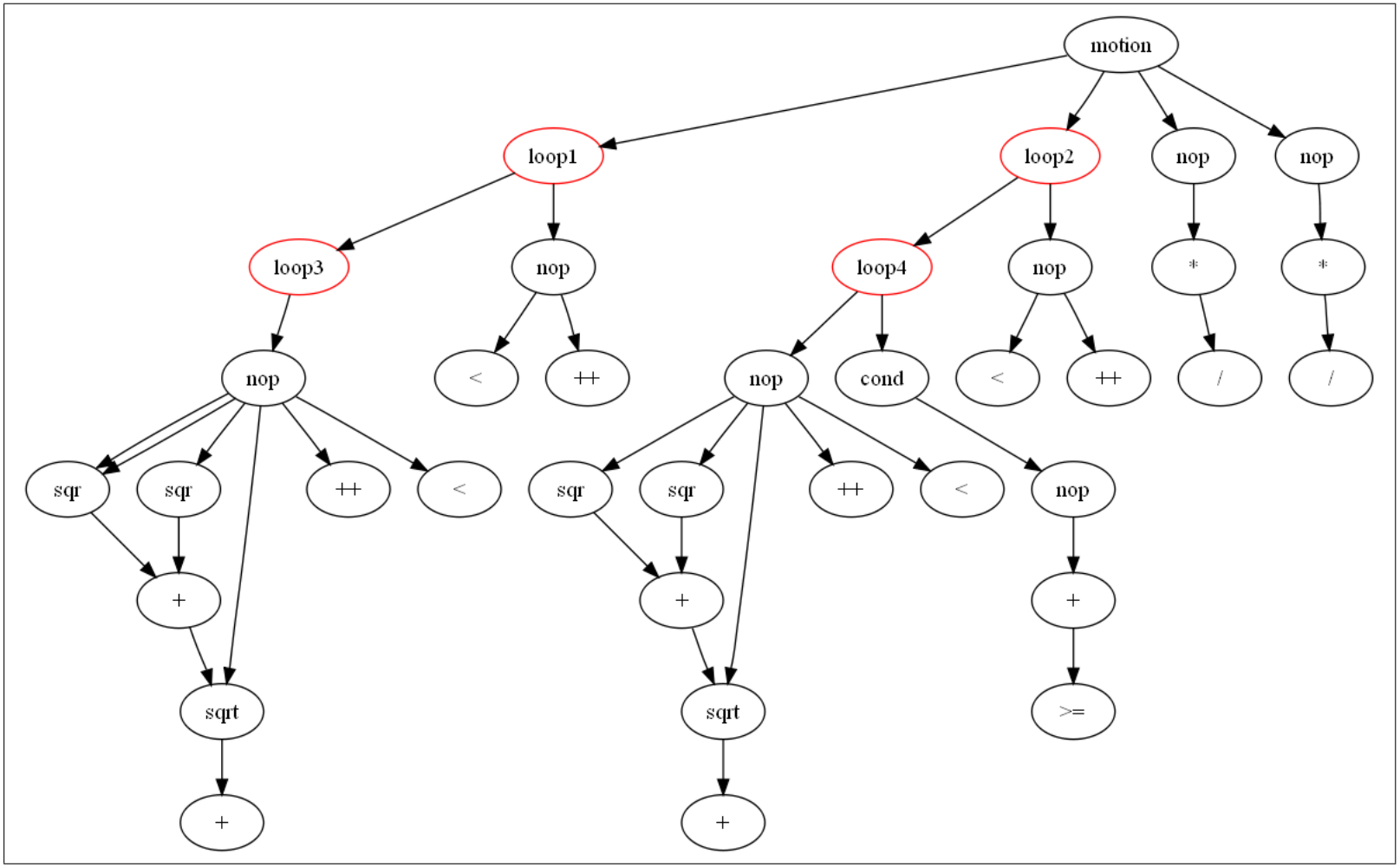

Figure 3.4. Motion intensity calculator CDFG

Table 3.4. Experiment results for motion intensity

	resultant time	operators chosen
Run Instance 1	28844 *timeunits*	sqr(PE)
Run Instance 2	30284 *timeunits*	sqr(PE)
Run Instance 3	22796 *timeunits*	Figure 3.5

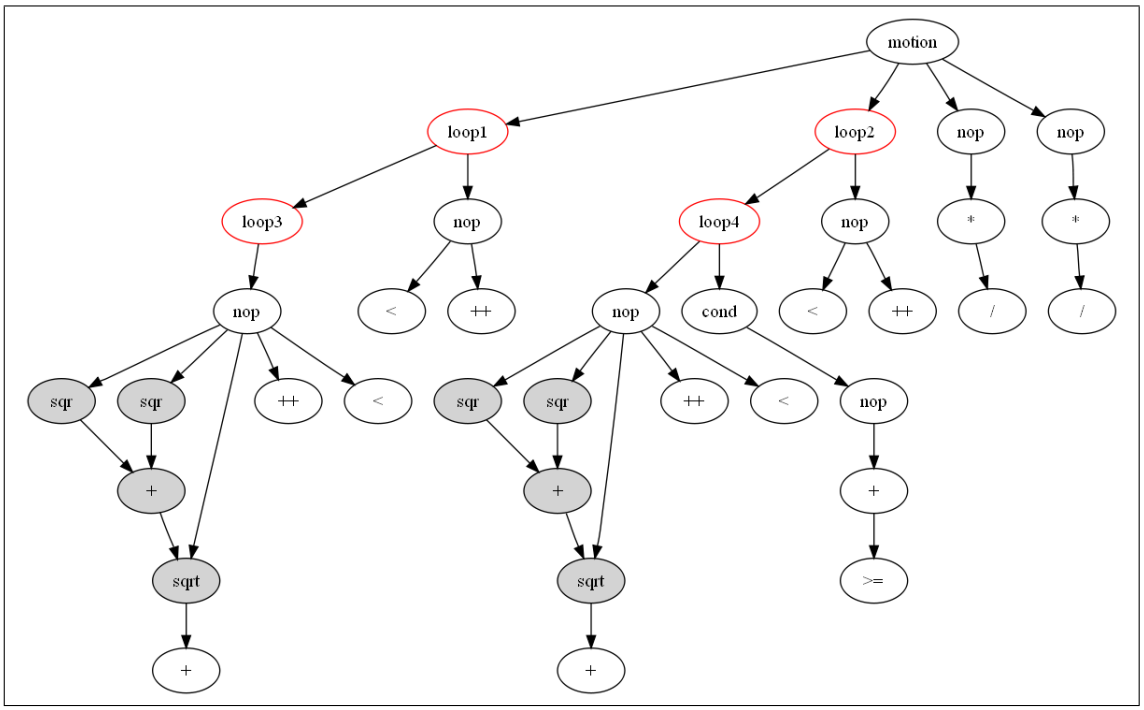

Figure 3.5. Motion Intensity CI selection: critic area system, special node: sqrt IP core (90/400 *hwtime*/*swtime*)

4. CONCLUSIONS

With the advent of customization of embedded processors, automation in customization research area has emerged. Automation is necessary in order the design inexpensive systems that meet quick-to-market deadlines. The research aims to fully automate the customization process, however it is directed with user input to some extent for satisfactory results.

Generality of the applications is another target to work on because embedded systems with specialized applications exhibit a highly varying nature. The wide range of the attributes that the applications possess can be exemplified like parallelism in matrix calculations in multimedia applications, recurrency of operations in cryptography applications, hard time constraints of real-time-systems... The embedded systems also has to be customized according to different system attributes like being area-critic or time-critic etc. If generality is targeted in tailoring the processor to an application, the varying nature of the applications and systems should be evaluated by the customization process.

In GAIA Explorer, an expert system, GES, at the intention of assisting the customization process has been implemented in this thesis. The imprecise data are represented with fuzzy logic. The customization domain knowledge is constructed into expert system with rules. Expert system is augmented with a graph evaluater, GGE, to assess the topological attributes of the nodes such as recurrency of operations, parallelism, control inherited behaviour of nodes and data dependency flow. The control of the GAIA explorer is handled by GGE.

The abstraction of the data nodes results in flexible partitioning process. The input can be as fine-grained as assembly level, as coarse-grained as user defined functions or a mixture of both. The flexibility results in IP cores to be evaluated as special nodes during partitioning process.

According to the results, the conflicting/cooperating attributes of the nodes have been evaluated in the expert system. GES, acting as a voter for customization process, marks the HW/PE membership of the nodes. The voting results are passed to the GGE. Selection is performed until area constraint is violated without changing the choice order of the GES.

However in node selection phase, GGE intervenes to the selection process. If a node in a recurring pattern, has a high membership, GGE selects the other nodes in the pattern for node fusion, exploiting recurrent nature of the application. A naive approach for choosing the pattern has been implemented and can further be improved with a method like graph covering. The methods used are given in section 2.2.6.

Node fusion is not favored other than patterns in GAIA Explorer so as not to jeopardize a high percentage HW node to be left in GPP due to area budget. Fusing the nodes around the selected nodes can result in area consumption by nodes that have low percentage HW attribute. However, when the data nodes are fine grained as in assembly level, fragmentation of the graph occurred. A control mechanism to prevent external fragmentation of the graph for fine grained data input is observed to be necessary.

In addition to the task of representing imprecise terms and constructing domain knowledge in an expert system, a recurrent isomorphic subgraph approach has been implemented. A rule in GES acts on nodes that are in pattern. In order to provide the relevant data to GES, GGE marks the nodes in pattern before passing them to GES. The solution has been achieved by mapping the recurrent subgraph problem to recurrent substring problem. A pseudo polynomial-time complexity algorithm used in multimedia applications to find the *longest subsequence* is modified in order to find the common substrings. In order to map the subgraph problem to substring problem, linearization of the graph has to be performed without losing data topology and data dependency of the nodes. A pool of strings are constructed from the nodes. Search space has been bound by a parameter depth-to-go. An enhanced approach to linearization of the graph can further be found to decrease the search space, eliminating

depth-to-go parameter.

For future directions, a user modifiable expert system can be implemented, to increase the generality and variety of the domain. The results show that a voting mechanism for conflicting and cooperating attributes of the nodes is achievable and logical. Since the attributes are discrete, new components can be added to the GGE along additional rules for GES depending on further requirements of different industries. Also a genetic algorithm can be employed for a self-learning system.

REFERENCES

1. P. Ienne, R. L., *Customizable Embedded Processors*, Morgan Kaufman Publishers, 2007.

2. Pan Yu, T. M., "Characterizing embedded applications for instruction-set extensible processors", *In Proc. of DAC*, 2004.

3. Pan Yu, T. M., "Scalable custom instructions identification for instruction-set extensible processors", *In Proc. of CASES*, 2004.

4. Yu, P. and T. Mitra, "Satisfying real-time constraints with custom instructions", pp. 166–171, 2005.

5. Togawa, N., K. Tachikake, Y. Miyaoka, M. Yanagisawa and T. Ohtsuki, "Instruction set and functional unit synthesis for SIMD processor cores", pp. 743–750, 2004.

6. K. Tachikake, Y. M. J. C. M. Y. T. O., N.Togawa, "REDEFIS A System with a Redefinable Instruction Set Processor", *In Proc. SBCCI*, 2006.

7. Arnold, M. and H. Corporaal, "Designing domain-specific processors", pp. 61–66, 2001.

8. "An algorithm for synthesis of large time-constrained heterogeneous adaptive systems", *ACM Trans. Des. Autom. Electron. Syst.*, Vol. 6, No. 2, pp. 207–225, 2001.

9. Knudsen, P. V. and J. Madsen, "PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning", p. 85, 1996.

10. Galuzzi, C., K. Bertels and S. Vassiliadis, "Graph Theory and Application Specific Processors", , November 2004.

11. K.Atasu, P., L.Pozzi, "Automatic Specific Instruction Set Extensions under Microarchitectural Constraints.", *In Proc. of DAC*, 2003.

12. X.Chen, Y., D.L. Maskell, "Fast Identification of Custom Instructions for Extensible Processors", , 2006.

13. Jason Cong, G. H. A. J. G. R. Z. Z., Yiping Fan, "Instruction set extension with shadow registers for configurable processors", *ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 99 –106, 2005.

14. Biswas, P., V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt, "Introduction of local memory elements in instruction set extensions", pp. 729–734, 2004.

15. Zou, G. and X. Liu, "An Efficient Approach to Custom Instruction Set Generation", pp. 547–550, 2005.

16. Vázquez, A., R. Dobrin, D. Sergi, J. P. Eckmann, Z. N. Oltvai and A. L. Barabási, "The topological relationship between the large-scale attributes and local interaction patterns of complex networks.", , December 2004.

17. Liem, C., T. May and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation", *European Design and Test Conference, 1994. EUROASIC, The European Event in ASIC Design, Proceedings.*, pp. 31–37, 28 Feb-3 Mar 1994.

18. Galuzzi, C., E. M. Panainte, Y. Yankova, K. Bertels and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions", pp. 160–165, 2006, `http://doi.acm.org/10.1145/1176254.1176293`.

19. C. M. Hohenauer, H. S., "Retargetable Code Optimization with SIMD Instructions", *In Proc. of CODE*, 2006.

20. F. Barat, P. B. G. D., M. Jayapala, "Software Pipelining for Coarse Grained Reconfigurable Instruction Set Processors.", *In Proc. of VLSID*, 2002.

21. Sonmez, N., *SIXD:A Configurable and Customizable SISD/SIMD Microprocessor Soft Core*, Master's thesis, Bogazici University, 2006.

22. Aditya, S., B. R. Rau and V. Kathail, "Automatic Architectural Synthesis of VLIW and EPIC Processors", pp. 107–113, 1999, `citeseer.ist.psu.edu/562518.html`.

23. J. Lee, N. D., K. Choi, "An Algorithm for Mapping Loops Onto Coarse-Grained Reconfigurable Architectures", *In Proc. of LCTES*, 2003.

24. Biswas, P., N. Dutt, P. Ienne and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage", pp. 212–217, 2006.

25. Catania, V., M. Malgeri and M. Russo, "Applying Fuzzy Logic to Codesign Partitioning", *IEEE Micro*, Vol. 17, No. 3, pp. 62–70, 1997.

26. Basu, R. S. M. A., "Knowledge representation in MICKEY: an expert system for designingmicroprocessor-based systems", *Systems, Man and Cybernetics*, 1997.

27. López, M. L., C. A. Iglesias and J. C. López, "A knowledge-based system for hardware-software partitioning", pp. 914–915, 1998.

28. Orchard, B., *FuzzyClips 10d Manual*, `http://iit-iti.nrc-cnrc.gc.ca/`.

29. "A Tool for Building Expert Systems http://clipsrules.sourceforge.net/", .

30. "Eclipse C/C++ Development", `http://www.eclipse.org/cdt/`.

31. "MINGW - http://www.mingw.org/", .

32. NASA, *Clips Basic Programming Guide*, `http://www.ghg.net/clips/CLIPS.html`.

33. Jeremy G.Siek, A. L., Lie-Quan Lee, *The Boost Grah Library*, Addison Wesley, 2001, `www.boost.org`.

34. "Boost Library", `http://www.boost.org/doc/libs`.

35. "Ratcliff/Obershelp Algorithm -http://www.ddj.com/184407970?pgno=5", .

36. Gonzalez, R. E., "Xtensa — A Configurable and Extensible Processor", *IEEE Micro*, Vol. 20, No. 2, pp. 60–70, /2000.

37. Quinn, M. J., *Parallel Programming in C with MPI and OpenMp*, McGrawHill, 2004.