A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units

Bhuvan Middha
bhuvan@cse.iitd.ernet.in

Varun Raj
varun@cse.iitd.ernet.in

Anup Gangwar
anup@cse.iitd.ernet.in

Anshul Kumar
anshul@cse.iitd.ernet.in

M. Balakrishnan
mbala@cse.iitd.ernet.in

Department of Computer Science and Engineering
Indian Institute of Technology Delhi, India

Paolo Ienne
Paolo.Ienne@epfl.ch

Processor Architecture Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland

ABSTRACT

It is widely accepted that use of an Application Specific Instruction Set Processor (ASIP) in an embedded system can provide a solution which is much more flexible than ASICs and much more efficient than standard processors in terms of performance and power consumption. However a lack of an acceptable design methodology and supporting tools for ASIPs limits their use even today. We present in this paper a methodology for design space exploration of high performance VLIW ASIPs by modeling Application Specific Functional Units in Trimaran Compiler Infrastructure. To demonstrate the effectiveness of our strategy we consider two important applications **FFT** and **Kalman Filter** and perform compute intensive operations in these applications via special Functional Units. The results we obtain are very promising with up to 2× speed improvement.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: VLIW architectures

General Terms

Performance

Keywords

Trimaran, VLIW, Performance, ASIP, Design Space Exploration

ISSS'02, October 2–4, 2002, Kyoto, Japan.

1. INTRODUCTION AND MOTIVATION

With the cost of silicon area decreasing, it is becoming more and more attractive to trade-off this area for the increased flexibility which an ASIP can provide. However, design methodologies and tools do not exist which can deliver ASIP designs suitable for embedded systems with challenging demands on performance. The work reported in this paper is aimed at developing a methodology for design space exploration and synthesis of high performance ASIPs.

In order to deliver high performance, an ASIP must exploit the instruction level parallelism (ILP) available in the given application. This points to VLIW architecture as a possible choice because it offers a better possibility of customization [1]. The number of Functional Units (FUs) and their organization into clusters (known as *clustered VLIW architecture*) is one dimension of architectural space which has already been explored [3]. The types of FUs and introduction of application specific FUs in the architecture is a relatively less explored dimension. Hence we consider a VLIW architecture which consists of a core set of FUs augmented with application specific coarse grain FUs. Specializing or customizing FUs for operations (or group of operations) occurring in a given application can potentially lead to high performance gains because of the following considerations [4]: (a) If the operands of an operation have a limited resolution (bit width), FU hardware can be simplified and made faster. (b) By chaining a sequence of operations in an FU, the computation time can be reduced. (c) Concurrent operations within a group can be more easily parallelized than parallelization across the FUs. Further, by mapping a group of operations to an FU, access to register file for the intermediate results is avoided. This reduction in *register pressure* has a beneficial effect on performance.

In order to evaluate the performance of an architecture for a specific application, one can follow an estimation based approach or a simulation based approach. The estimation based approach yields quick, but inaccurate results. Hence

for fine grain performance comparison of various architectures one needs to simulate the architecture running the code of the application. For VLIW processors, this is impractical without a compiler. To support architecture exploration, the compiler as well as the simulator need to be retargetable. Trimaran [5] though limited in terms of architectural space provides such tools.

In this paper, we present a framework built around the Trimaran infrastructure, which allows us to study the effect of application specific FUs on performance. We assume an executable specification in C and suggest a methodology to observe the effect of putting hardware accelerators for speeding up specific portions of the code. This paper is organized as follows: Section 2 describes the space of target architectures, Section 3, describes the performance evaluation framework, Section 4 describes the extensions to Trimaran for introduction of application specific FUs, Section 5 describes the case studies for validation of the work and finally Section 6 summarizes this work and also discusses the possible future directions.

2. SPACE OF TARGET ARCHITECTURES

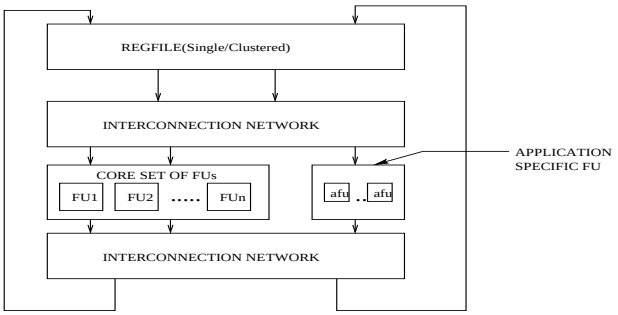

Figure 1: A typical VLIW ASIP Architecture

The target architecture which we consider for ASIP synthesis is as shown in Figure 1. It is essentially a VLIW processor with a core to support the usual fine grain operations like add, multiply, compare etc., augmented by application specific extensions. These extensions are centered around some medium or coarse grain FUs defining new instructions for implementing some critical functionality of the specific application. Actually the fine grain core may not be absolutely rigid, but generic in some limited sense. Further, it provides a default set of resources which are adequate to implement any part of the application.

Our focus in this paper is on exploring the use of coarse grain FUs for obtaining high performance ASIP architectures. We now look at the spectrum of custom FUs which we consider in the design space exploration. At one end of the spectrum there are multiple input single output units without any memory accesses and control, termed as **MISOs** [6]. This is the simplest generalization of basic fine grain operations which typically take one or two inputs and produce one result. The next conceivable generalization is to allow multiple outputs to be produced by an FU, making it a **MIMO** or a multiple input multiple output unit. The cycles in which various operands of a MIMO are input and results are output, relative to the beginning cycle, define the *I/O time shape* of such a MIMO [7]. If the cycles in which I/O occurs are fixed, the time shape is considered to be *rigid*. On the other hand, if the I/O operations are hold-able, that is the cycles in which they occur could be delayed, the time shape is said to be *flexible* which eases scheduling. A Basic MIMO takes all its operands from register files, therefore a further extension can be in terms of considering **MIMOs with load/store** which are capable of accessing the memory. An orthogonal dimension is whether conditionals are permitted within a FU or not. This further enhances its scope but it causes the latency of the FU to be variable. Such variable latency makes pure VLIW kind of scheduling difficult. One can even think of mapping loops to an FU with even the loop control inside the FU but again this will require some run time control and synchronization, e.g. handshaking.

Name	Inputs and Sources	Outputs and Dests.	I/O Policy
MISO	Multiple (Regfile)	Single (Regfile)	Flexible or Rigid
MIMO	Multiple (Regfile)	Multiple (Regfile)	Flexible or Rigid
MIMO with LD/ST	Multiple (Regfile or Memory)	Multiple (Regfile or Memory)	Flexible or Rigid for Regfiles, Block LD/ST at beginning and end of operation

Table 1: Architectural spectrum of custom FUs

3. FRAMEWORK FOR PERFORMANCE EVALUATION

Here we consider **Trimaran Compiler Infrastructure** as the framework for performance evaluation. The Trimaran system is based on the **HPL-PD** architecture which is a parameteric processor architecture conceived for research in instruction-level parallelism. The HPL-PD opcode repertoire, at its core, is similar to that of a RISC-like load/store architecture, with standard integer, floating point (including fused multiply-add type of operations) and memory operations. We map the core part of our target architecture to the HPL-PD architecture.

The Trimaran compiler infrastructure, as shown in Figure 2, consists of a compiler front-end, IMPACT, compiler back-end, Elcor [2], and a simulator generator. The framework is parameterized using a machine description facility, HMDES [10]. We briefly describe each of these tools.

The IMPACT compiler system, is used by the Trimaran system as its front end. This front-end performs, ANSI C parsing, code profiling, classical code optimizations along with block formation.

The High Level Machine Description Facility or HMDES is the machine description language used in Trimaran system. This language describes a processor architecture from the compiler's point of view. To this end it specifies the *instruction format*, *resource usages and reservation tables*, *latency information*, *operation information* and *some compiler specific information*. The *instruction format* conveys what operands are allowed by each type of operation, *resource usages* specify how operations use processors resources as they execute and *latency information* specifies how to calculate dependence distances between operations. Finally, *operation information* specifies the operations sup-

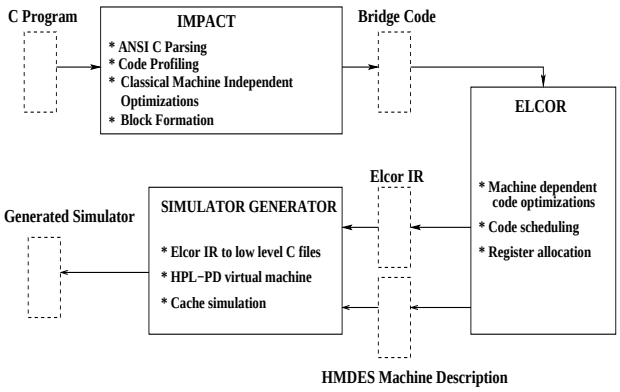

Figure 2: The Trimaran Compiler Infrastructure

ported by the architecture. and describes each of them in terms of there *Scheduling Alternatives* which includes the format, resource usage and latency.

Elcor is Trimaran's back-end for the HPL-PD architecture. It performs three tasks: (a) code selection and scheduling. (b) register allocation. (c) machine dependent code optimizations. Elcor is parameterized by the machine description facility to a large extent. As shown in Figure 2, it takes as input the bridge code produced by front-end along with a HMDES machine specification and produces an Elcor IR file. The IR is annotated with HPL-PD assembly instructions. The internal representation of **Elcor IR** consists of a set of C++ objects. All optimization modules in the Elcor IR use the interface provided by these objects to carry out optimizations. Optimizations are simply IR to IR transformations.

The Trimaran framework also consists of a **simulator** which is used to generate various statistics such as compute cycles, total number of operations, etc.

The limitations of the Trimaran framework are that firstly, it is built around the HPL-PD architectural domain. Hence, it only supports operations which are a subset of HPL-PD operations. Secondly, the Trimaran framework does not completely support clustered VLIW architecture. It has a single register file of each type (e.g. integer regfile, floating point regfile etc). Each integer FU accesses the same integer regfile. Hence, we cannot evaluate performance for clustered architectures.

4. EXTENDING TRIMARAN INFRASTRUCTURE

In this section, we consider the problem of introducing coarse grain FUs in a compiler infrastructure. We assume that the compiler infrastructure consists of a machine description facility, a compiler front and back end and a retargetable simulator.

The first step involves defining a new machine operation O and a new resource R in the system. The operation O will be performed by R which corresponds to a coarse grain FU in the architecture. The operation O will be defined in terms of the operation format, operation latency and the resource usage. After this the compiler needs to be modified so that it is able to generate code for this new operation. For this one requires a retargetable compiler parameterized with the machine description. The front end should be able to identify the pattern corresponding to O in the application source code and emit the appropriate Intermediate Representation (IR). The back end should be able to generate code corresponding to this IR. The former is in general a very hard problem as all the information cannot be coded in the machine description, which will enable the front end to identify the pattern in the source code corresponding to coarse operation O.

Another approach could be that the front end remains unchanged. The application code is itself modified so that the desired computation (to be carried by coarse grain FU) is replaced by an *external function call*. The IR will consist of nodes corresponding to this function call. Then one can modify the IR itself to replace these nodes by a new node corresponding to the operation O. The back end will then treat this node as any other standard machine operation (e.g. ADD) and generate code for it. Finally one needs to define the operation semantics inside the retargetable simulator so that various statistics can be generated.

We take the latter approach to extend the Trimaran infrastructure. Each new operation is represented in terms of an external function call. The function name and coarse grain FU binding is implicit. The function name itself specifies to which FU it should be bound. We identify the function call in the IR of the code and replace it with a coarse grain operation in the IR. There is a one to one mapping between the coarse grain operation and the name of the function in the application code. The operation now propagates through the whole suite of optimizations done by the compiler. We also define the semantics of the new operation in the Trimaran simulator.

The Trimaran framework has no notion of register file ports. It assumes each regfile to have an unlimited number of ports. We incorporate the notion of regfile ports in the framework with parameterized number of read/write ports corresponding to each regfile. This is an essential constraint in VLIW ASIP design as access time, area and power consumption sharply increase with the number of ports in a register file. The modified framework is shown in Figure 3. The shaded portion represents those parts of the framework which have been modified, with changes indicated along with each part. We have successfully modeled the three classes of FUs described above, MISOs, Basic MIMOs and MIMOs with LD/ST. In the following paragraphs we describe each of these.

4.1 Modeling MISOs

We have identified and successfully tested the following approach for introducing MISOs in the Trimaran Compiler Infrastructure. The application program in C consists of a prototype declaration of a function which the user wants to perform via a special functional unit. This is illustrated with the help of an example :

```
main(){
 int a,b,c,d;
 a=3; b=4; c=5;
 //The following computation
 //is to be done via special FU
 d = (a+b)*(b+c)*(a+c);
}
```

Let us define a new functional unit which takes in 3 inputs a, b and c and produces one output d. For defining the new

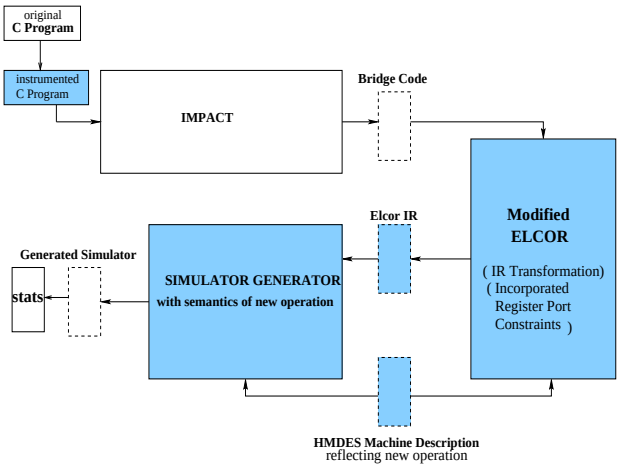

Figure 3: Modified Trimaran Framework

functional unit we declare a prototype function in C; i.e., we do not define its functionality but only declare its interface.

```
int miso_fun(int a, int b, int c) ;
main(){
 int a,b,c,d;
 a=3; b=4; c=5;
 d = miso_fun(a,b,c);
}
```

Since the function is not completely defined inside the application, after passing through the front end it appears in the form of an *external function call* in the Trimaran bridge code along with the relevant annotations which consists of name of the function etc. After first pass through Elcor it appears in the form of two Elcor Operations *PBRR*(prepare to branch) and *BRL*(branch and link). We identify this combination of *PBRR* and *BRL* corresponding to the prototype function in the IR and replace this combination by a new node in the IR which corresponds to a new Elcor Operation and represents the FU which we want to introduce. The source and destination operands of this new Elcor Operation are the same as the source and destination operands of the prototype function call.

Finally the semantics of the new operation are defined in the simulator which involves defining destination as a function of sources. As illustrated in Figure 4, we consider an

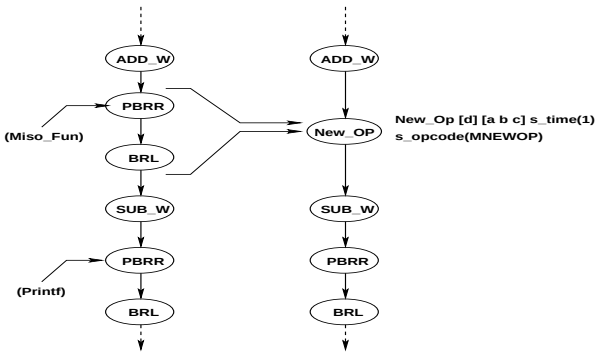

Figure 4: Modifications in Elcor IR

example of a part of IR in which the PBRR and BRL operations corresponding to miso_fun function call are replaced by NEW_OP and other such combinations remain unchanged. The new operation is also defined in MDES (with opcode MNEWOP) which involves defining its Operation Format , Number of Resources (FUs), Operation Latency, Resource Usage and the Reservation Table.

4.2 Modeling MIMOs

Since a function cannot return more than 1 value by definition, we take a slightly different approach here. Instead of a prototype function returning a value we consider a void function. We reserve some registers (through the C code itself, by giving some compiler directives) and the function returns values in those registers. This is illustrated by the following example:

```
main(){
 int a,b,c,d;
 a=4; b=5;
 //The following  computations
 //are to be done via special FU
 c = a+b;
 d = a-b;
 printf(c,d);
}
```

Let us define a new functional unit which takes in 2 inputs a and b and return outputs (a+b) and (a-b). Hence it is a MIMO. The application consists of void prototype function declaration.

```
void mimo_fun(int a, int b);
main(){
 int a,b, ret1,ret2;
 //Some code and directives
 //to reserve registers
 ........
 mimo_fun(a,b);
 //Values will be returned in ret1 & ret2
 printf(ret1,ret2);
}
```

As in the previous case the prototype function call appears as a combination of *PBRR* and *BRL* in the Elcor IR. But now in addition to replacing the above combination by a new Elcor Operation we set the destinations of operation as the registers reserved for this purpose (that is registers corresponding to variables ret1 and ret2 in the above example). The operation is also defined in MDES and its semantics are defined in the simulator.

4.3 Modeling MIMOs with load/store

To handle **MIMOs** with capability of interaction with memory we make modifications only in the MDES. Basically in the reservation table corresponding to the operation we also reserve memory units in each time unit where interaction with the memory is required. We have multiple memory units in the system, so one of the units is reserved for performing this operation while others can handle normal load/store operations. The architecture assumes each LD/ST unit has ports to memory so they can be active simultaneously. While making the function call we also pass the addresses of the memory locations from which data is required. In the first few cycles of the operation memory resident data is accessed with the help of LD/ST unit and stored

in local buffers. Then the computation is performed and finally data is written into the memory, if required, again with the help of LD/ST unit. The semantics are handled in the simulator in a similar way as for basic MIMOs.

4.4 Imposing Register Port Constraints

The Trimaran framework has no notion of regfile ports. It primarily has 1 regfile of each type (GPR, control regfile, floating point regfile, branch target and predicate regfile). The scheduler, for example can schedule any number of integer operations in parallel depending on the availability of resources. This implies each regfile has infinite number of ports in theory. So we impose these port constraints in the architectural framework because all the special FUs like **MIMOs** will have a large no. of sources and many destinations. To incorporate these constraints we build a *Time X Regport* table for read as well as write ports in which at each instant of time corresponding to each regfile the utilization of its read/write ports is maintained. Before scheduling any operation the table is checked for availability of read/write ports along with the availability of resources. In the *rigid I/O time shape* model if a particular FU has more sources than the number of read ports or more destinations than the number of write ports then the ports are reserved in the very next time instant, i.e., the cycles in which I/O occurs is fixed.

In the *flexible I/O time shape* model an FU is divided into various stages and the number of sources and destinations in each stage lie within the maximum number of read/write ports. A *flow dependency edge* is added between each stage to ensure each stage is scheduled after its predecessors are scheduled depending on the availability of resources and ports. But it is flexible in the sense that stage *i* can be scheduled *anytime* after stage *i-1* has been scheduled. This is shown in Figure 5.

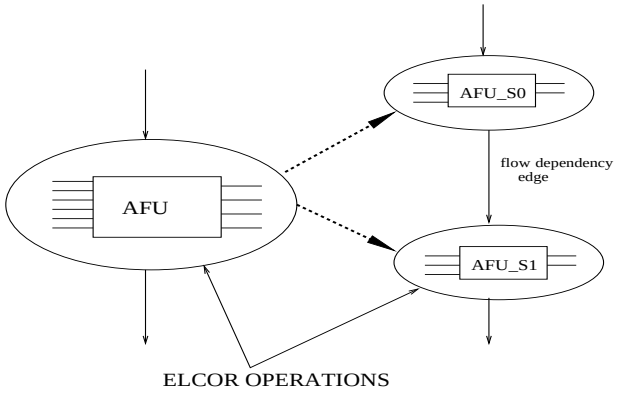

Figure 5: Flexible I/O Time shape

5. CASE STUDIES

5.1 Fast Fourier Transform (FFT)

To illustrate the concept of MISOs and MIMOs and to evaluate the performance gain when special functional units are present in the system we consider a standard N-point FFT application. FFT forms the heart of many image transformation packages, thus is an interesting application to consider speed up. The heart of FFT is the repetitive computation of the **butterfly** operation, The butterfly operation is shown in Figure 6. In this application we replace the butterfly operation which has 6 sources (as each of the 3 sources are complex numbers) and 4 destinations with a Multiple Input Multiple Output FU. The model conforms to the *rigid I/O time shape model* with each regfile having 4 read ports and 2 write ports. The latency of the butterfly operation is set to 8.

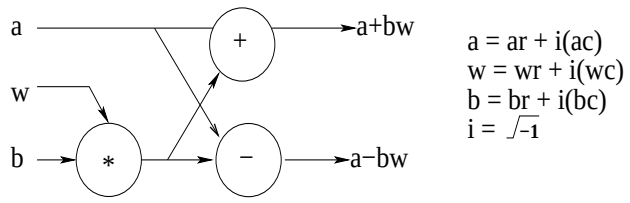

Figure 6: Butterfly Operation

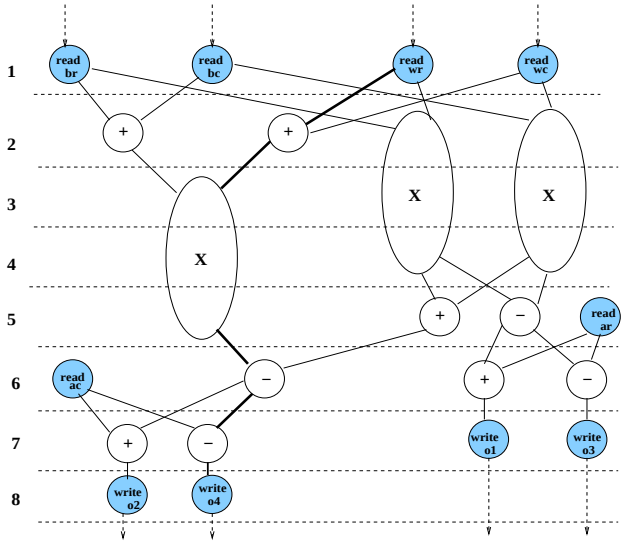

Figure 7: Data flow graph of butterfly operation

We implement the DFG shown in Figure 7 in the 6-4 MIMO corresponding to the butterfly operation. The highlighted portion represents one of the many critical paths. The length of the critical path is 8, assuming 4 reads and 2 writes are permitted in each cycle. The shaded portions represent the read and write operations. The latency of the multiplication operation in the base architecture is 3, whilst that of arithmetic operations is 1. The results of introduction of the new functional unit are shown in Table 2.

n-point FFT	Without Special FU (cycles)	With Special FU (cycles)
2	236	209
4	549	341
8	1209	583
16	2649	1081

Table 2: Compute cycles for varying n in FFT

Table 2 shows that as the value of n increases the number of butterfly operations also increase and therefore perfor-

mance gain also increases. For n=16 the speed improvement is almost 2.5×.

We have also implemented the *flexible I/O timeshape* model corresponding to the FFT application. The results obtained are similar to as in the rigid case.

5.2 Kalman Filter

The Kalman filter [8] is a set of mathematical equations that provides an efficient computational (recursive) solution of the least-squares method. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown. The Kalman filter basically consists of two main functions *predict_state* which predicts the state of the system and *kalman_update* which updates the system. We built special FUs to perform various frequently occurring operations in these functions. Many of these operations involve manipulation of various arrays which involved handling memory within the AFU. In all, we introduced 5 *MISOs with load/store* to handle the various operations. The description of each AFU is shown in Table 3. The semantics are specified in the form of destination as a function of sources, where s_i represents the i^{th} source of the AFU an d_i represents the i^{th} destination of the AFU. The latency of each FU conformed to the amount of

AFU No.	No. of Inputs	No. of outputs	Semantics
1	5	1	$d_1=(s_1+(s_2*(s_3+s_4+s_5*s_2)))$
2	3	1	$d_1=(s_1+(s_2*s_3))$
3	5	1	$d_1=(-s_1*s_2+s_3*s_4)/s_5$
4	5	1	$d_1=(s_1+s_2*s_3+s_4*s_5)$
5	5	1	$d_1=(s_1-s_2*s_3-s_4*s_5)$

Table 3: Kalman Filter AFUs

computation involved and the model followed was *rigid I/O time shape model*. The number of read ports in each regfile were 3 and number of write ports were 2.

The results obtained are shown in Table 4.

Function	Without Special FU(cycles)	With Special FU(cycles)
Predict_State	699	498
Kalman_Update	774	342

Table 4: Kalman Filter Results

We have used 3 special FUs in the *kalman_update* function and 2 special FUs in the *predict_state* function, the performance is better in the case of latter because there were more operations that could be mapped to these special FUs than the former case. As can be observed from Table 4, the number of cycles have come down to less than half in the predict_state function, which implies a fairly large performance gain.

6. CONCLUSION AND FUTURE WORK

We have presented a framework to quickly evaluate the performance gain obtained when special application specific functional units are introduced in the architecture. The framework can be used for extensive design space exploration and the user can experiment by mapping various compute intensive parts of the application to special FUs in hardware and comparing the relative performance estimates under accurate implementation constraints. This can ease the synthesis of ASIPs corresponding to these set of applications. Research is going on in the area of automatic topology based identification of instruction set extensions for embedded processors [9]. Currently the potential candidates are identified manually. A possible future work can be to create an automatic identification-evaluation framework, which automatically identifies the potential candidates based on some speedup factors associated with each instruction and then evaluates them using this extended Trimaran framework for possible gains. The modeling does not take into account the relative cost of the special FU. A possible extension can be an introduction of cost model which evaluates the area corresponding to each special FU so that one can even evaluate the performance-area trade off. Besides, the FUs are of limited complexity, one can extend the framework to introduce FUs which are capable of handling conditionals and loops also. Currently,the changes required in the Trimaran framework for the introduction of any particular FU are manual. We are making efforts to automate it to the extent possible.

7. REFERENCES

[1] Shail Aditya, B.R. Rau and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings of 12th ISSS*. November, 1999.

[2] Shail Aditya, Vinod Kathail, and B. Ramakrishna Rau. Elcor's Machine Description System: Version 3.0. Technical Report HPL-1998-128, Hewlett-Packard Laboratories, October 1998.

[3] Margarida F. Jacome et al. Clustered VLIW architectures with predicated switching. In *DAC*, pages 696-701,2001.

[4] Paolo Ienne, Laura Pozzi, M. Vuletic. On the Limits of Processor Specialisation by Mapping Dataflow Sections on Ad-hoc Functional Units. CS Technical Report 01/376, LAP, EPFL, Lausanne. December 2001.

[5] The Trimaran Compiler Infrastructure, http://www.trimaran.org.

[6] Cesare Alippi et al. A DAG based design approach for reconfigurable VLIW processors. In *Proceedings of the DATE*, pages 778-79, March 1999.

[7] N.G. Busa et al. Scheduling coarse grain operations for VLIW processors In *Proceedings of the 13th ISSS*, pages 47-53, Madrid, September 2000.

[8] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical Report, Department of Comp. Sc. and Engg., Univ. of North Carolina at Chapel Hill, March 2002.

[9] Pozzi, Laura and Vuletić, Miljan and Ienne, Paolo. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Paris, March 2002.

[10] J. Gyllenhaal, B. Rau, and W. Hwu. HMDES version 2.0 specification, IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1996.