Design Space Exploration for a Custom VLIW Architecture

M.K.Jain, PhD.
Mohan Lal Sukhadia University
Department of Computer Science

Veena Ramnani
Mohan Lal Sukhadia University
Department of Computer Science

ABSTRACT
The increasing complexity of algorithms and embedded systems constraints has lead to advanced design methodologies. Hardware/Software co-design methodology has made it possible to find an optimal architecture for a given application by exploring the design space before building a real hardware prototype. The Design Space Exploration is basically exploring the various processor architectures in order to search for a processor architecture that satisfies different conflicting criteria such as chip area, speed, power consumption or on-chip memory requirements. The output is a set of different architectures representing the different tradeoffs. Retargetable compiler is an important tool in design space exploration (DSE). Retargetable compiler is capable of generating code for different target processors, by reusing most of the code. The objective of this research is to develop a retargetable compiler that can generate efficient code in terms of code size, cycle count and retargetability efforts for a VLIW processor.

General Terms
Embedded Systems, Design Space Exploration (DSE)

Keywords
Retargetable Compilers, VLIW, ILP

1. INTRODUCTION
Modern system-level design libraries frequently consists of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIWs and hybrid ASIPs. Embedded systems facilitate easy re-design of processor-memory based systems. The designer can incorporate modifications in the behavior and operation aspect of the architecture late in the design stage. ASIP are a compromise between the non-programmable ASICs and general purpose processors (GPP). ASIP design [1] [2] [3] allows a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application. The ASIP designer is faced with the task of rapidly exploring and evaluating different architectural and memory configurations. Furthermore, shrinking time-to-market has created an urgent need to automatically generate compiler/simulator tool-kit. There are two approaches for performance estimation using ASIP design: scheduler based approach and simulator based approach.
Scheduler Based Approach: In scheduler based approach the problem is formulated as a resource constrained scheduling problem. The application is scheduled to generate an estimate of the cycle count. ***Simulator Based Approach:*** A retargetable compiler is constructed for each architecture to be evaluated. This compiler is used to generate code. This generated code is given as input to a retargetable simulator which is also designed for the same architecture under evaluation. This simulator generates the performance estimates and other statistics.

In their research work, the authors have used the simulator based approach. The objective of the research is to design a reatrgetable compiler for various processor architectures.

2. RETARGETABLE COMPILERS
Retargetable compilers are a promising approach for automatic compiler generation. A compiler is said to be"retargetable‟ if it can be used to generate code for different processor architectures by reusing significant compiler source code. This has resulted in a paradigm shift towards a language-based design methodology using Architecture Description Language (ADL) for embedded System-on-Chip (SOC) optimization, exploration of architecture /compiler co-designs and automatic compiler/simulator generation. However, whatever approach is used, the performance depends on the back end of the compiler i.e. instruction selection, register allocation and instruction scheduling.

As the contemporary applications for embedded system become more and more resource intensive and demand fast execution time, it has become imperative to design new architectures for them. Sequential has given way to parallel, as the parallel execution is supposed to use the hardware fully and have faster resulting execution. In this context, Very Long Instruction Word (VLIW) architecture is very useful in order to accelerate processing speed. It exploits the Instructional Level Parallelism (ILP) to result in better performances [4] [5].

This paper explains such exploration for VLIW architecture. We have taken VEX to be the base system for this paper. We have written a retargetable compiler that can generate code for a VLIW architecture similar to the one used in VEX. The code generated by our compiler is compared with the code generated by VEX and the performance statistics are compared.

The section 3 begins with a brief description of VLIW architecture and concepts. Then we present some descriptions about the VEX system: Instruction Set Architecture and VEX compiler. Section 4 illustrates our DSE methodology: architecture and parameter space definition and criteria used for each exploration iteration. Experiment results are shown in section 5. The conclusions and perspectives of our work are mentioned at the end in Section 6.

3. THE VEX SYSTEM

3.1. Concept of Very Long Instruction Word (VLIW) Architecture

Instruction Level Parallelism (ILP) is being used in high performance processors to achieve high execution speeds. Very Long Instruction Word (VLIW) is one of such approach to design processors with high levels of ILP by executing long instructions composed of multiple operations. In the VLIW architecture, the compiler has the responsibility for creating a group of operations that can be simultaneously issued [5]. The VLIW architecture does not make dynamically any decisions about multiple instruction issue and scheduling, and thus is efficient and fast.

In VLIW machine, the data path consists of multiple pipelined functional units, each of which can be independently controlled through dedicated fields in a *very long* instruction. The distinctive feature of VLIW architectures is that these long instructions are the machine instructions. The key to generating efficient code for the VLIW machine is global code compaction.

3.2. The VEX system: VLIW example

VEX [5] models a scalable platform to design embedded VLIW processors that allows variation in issue width, in the number of functional units and register, and in the processor instructions set. We can distinct the three following components for VEX system.

3.2.1. VEX ISA (Instruction Set Architecture)

VEX ISA consists of a flexible architecture, modeled of the family embedded cores HP/ST Lx [6]. The basic structure of this core (cluster) is given in Figure 1. By configuring, users can build a multi-clusters VLIW architecture that is customizable to individual application domains. The scalability gives the possibility to change the number of clusters, execution units, registers and latencies. With customizability, users can define specialized instructions. A multi-cluster implementation is given in Figure 2. VEX includes a complete exposure of all architecture latencies and resource constraints. It consists of parallel executions units, parallel memory pipelines, a large visible register set and an efficient branch architecture.

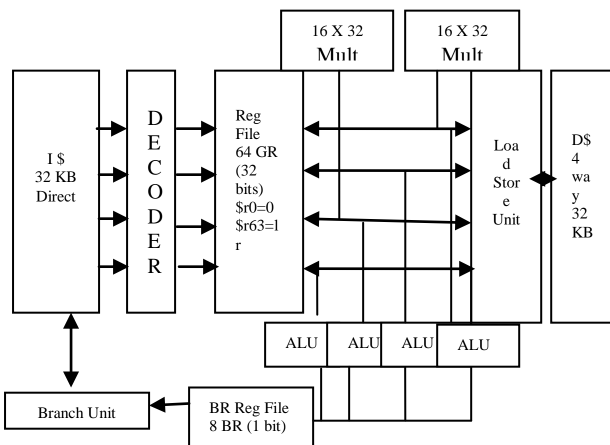

Figure 1:The default VEX cluster architecture

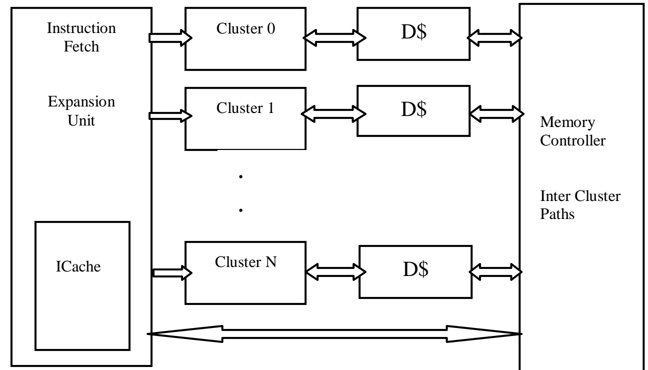

Figure 2: A multi-cluster implementation.

3.2.2 The VEX C Compiler

The VEX C compiler is a derivation of the Lx/ST200 C compiler, itself a descendant of the *Multiflow* C compiler. It is a robust, ISO/C89 compiler that uses *trace scheduling* as its global scheduling engine. A very flexible programmable machine model determines the target architecture. For VEX, we selectively expose some of the parameters to allow architecture exploration by changing the number of clusters, execution units, issue width, and operation latencies without having to recompile the compiler. Hewlett-Packard Laboratories provide a toolchain [7] with the compiler.

4. OUR DSE METHODOLOGY

Our compiler generates code for architecture similar to that of the VEX architecture. The architecture consists of four integer ALUs, two 16*32-bit multiply units and a data cache port. The compiler can issue up to four operations in an instruction. The register-set consists of 64general purpose 32-bit registers and eight 1-bit branch registers. In addition to the above resources, the architecture also contains a Control Unit (Branch Unit) for program sequencing.

For high performance to be achieved in superscalar and VLIW processors, the compiler must reorder the instructions in the program to make effective use of the instruction-level parallelism [8] offered by the processor, an optimization called *instruction scheduling*. A specialized form of this optimization, called *software pipelining*, is specifically designed to handle inner loops. Software pipelining is almost always applied in concert with loop unrolling, further increasing the potential for exploiting parallelism. Several different iterations can be overlapped and a wide set of instructions can execute the entire loop body in a single cycle.

The important point while generating the code for VLIW architecture is maximizing the ILP (Instruction Level Parallelism). Since, we are using VLIW architecture, it has multiple functional units and the generated code should optimally use these units. Our aim while generating the code was to include more and more operations in an instruction, keeping in mind their dependencies.

Another point, the authors have considered for code generation is code optimization. Loop unrolling is important in VLIW architecture. But, the code should not be unrolled completely. Unrolling the loop completely, increases the code size, which in turn increases the number of instructions and will reduce the ILP. In our compiler, we have not unrolled the loop completely, which has reduced the size of the code,

reduced the number of instructions and hence increased the ILP.

5. RESULTS

The authors have generated the code for the VLIW processor explained above. The glimpse of the code generated by VEX compiler and that generated by our compiler is given in the table below:

Code by VEX Compiler	Code by our compiler
.trace 1 c0 add $r0.1 = $r0.1, (-0x80) c0 mov $r0.3 = 2 ## 2(SI32) c0 mov $r0.2 = 1 ## 1(SI32) ;; ## 0 c0 add $r0.4 = $r0.1, 0x50 ## bblock 0, line 4, t1, t15, offset(x?1.2)=0x50(P32) c0 add $r0.5 = $r0.1, 0x28 ## bblock 0, line 4, t2, t15, offset(y?1.2)=0x28(P32) c0 mov $r0.6 = $r0.1 ## bblock 0, line 4, t3, t15 c0 mov $r0.7 = 3 ## 3(SI32) ;; ## 1 c0 mov $r0.11 = 7 ## 7(SI32) c0 mov $r0.10 = 6 ## 6(SI32) c0 mov $r0.9 = 5 ## 5(SI32) c0 mov $r0.8 = 4 ## 4(SI32) ;; ## 2 c0 stw 0[$r0.4] = $r0.0 ## bblock 0, line 8, t1, 0(SI32) c0 mov $r0.13 = 9 ## 9(SI32) c0 mov $r0.12 = 8 ## 8(SI32) ;; ## 3 c0 stw 0[$r0.5] = $r0.0 ## bblock 0, line 9, t2, 0(SI32) ;; ## 4 c0 stw 0[$r0.6] = $r0.0 ## bblock 0, line 10, t3, 0(SI32) ;; ## 5 c0 stw 4[$r0.4] = $r0.2 ## bblock 0, line 8-1, t1, 1(SI32)	.trace 1 c0 add $r0.1 = $r0.1, (-0x60) c0 mov $r0.4 = $r0.1 c0 add $r0.5 = $r0.1, 0x28 c0 add $r0.6 = $r0.1, 0x50 ;; c0 mov $r0.2 = 1 c0 mov $r0.3 = 2 c0 mov $r0.7 = 3 c0 mov $r0.8 = 4 ;; L0?3: c0 cmplt $b0.0 =$r0.0, 10 c0 brf $b0.0, L1?3 c0 stw 0[$r0.4] = $r0.0 ;; c0 stw 0[$r0.5] = $r0.0 ;; c0 stw 0[$r0.6] = $r0.0 c0 add $r0.0=$r0.0, 5 ;; c0 stw 4[$r0.4] = $r0.2 ;; c0 stw 4[$r0.5] = $r0.2 ;; c0 stw 4[$r0.6] = $r0.2 c0 add $r0.2=$r0.2, 5 ;; c0 stw 8[$r0.4] = $r0.3 ;; c0 stw 8[$r0.5] = $r0.3 ;; c0 stw 8[$r0.6] = $r0.3 c0 add $r0.3=$r0.3, 5 ;; c0 stw 12[$r0.4] = $r0.7 ;;
;; ## 6 c0 stw 4[$r0.5] = $r0.2 ## bblock 0, line 9-1, t2, 1(SI32) ;; ## 7 c0 stw 4[$r0.6] = $r0.2 ## bblock 0, line 10-1, t3, 1(SI32) ;; ## 8 c0 stw 8[$r0.4] = $r0.3 ## bblock 0, line 14, t1, 2(SI32) ;; ## 9 c0 stw 8[$r0.5] = $r0.3 ## bblock 0, line 9-2, t2, 2(SI32) ;; ## 10 c0 stw 8[$r0.6] = $r0.3 ## bblock 0, line 10-2, t3, 2(SI32) ;; ## 11 c0 stw 12[$r0.4] = $r0.7 ## bblock 0, line 14-1, t1, 3(SI32) ;; ## 12 c0 stw 12[$r0.5] = $r0.7 ## bblock 0, line 9-3, t2, 3(SI32) ;; ## 13 c0 stw 12[$r0.6] = $r0.7 ## bblock 0, line 10-3, t3, 3(SI32) ;; ## 14 c0 stw 16[$r0.4] = $r0.8 ## bblock 0, line 14-2, t1, 4(SI32) ;; ## 15 c0 stw 16[$r0.5] = $r0.8 ## bblock 0, line 9-4, t2, 4(SI32) ;; ## 16 c0 stw 16[$r0.6] = $r0.8 ## bblock 0, line 10-4, t3, 4(SI32) ;; ## 14 c0 stw 16[$r0.4] = $r0.8 ## bblock 0, line 14-2, t1, 4(SI32) ;; ## 15 c0 stw 16[$r0.5] = $r0.8 ## bblock 0, line 9-4, t2, 4(SI32) ;; ## 16	c0 stw 12[$r0.5] = $r0.7 ;; c0 stw 12[$r0.6] = $r0.7 c0 add $r0.7=$r0.7, 5 ;; c0 stw 16[$r0.4] = $r0.8 c0 add $r0.4=$r0.4, 20 ;; c0 stw 16[$r0.5] = $r0.8 c0 add $r0.5=$r0.5, 20 ;; c0 stw 16[$r0.6] = $r0.8 c0 add $r0.8=$r0.8, 5 c0 add $r0.6=$r0.6, 20 c0 goto L0?3 ;;

```
        c0    stw 16[$r0.6] =
$r0.8   ## bblock 0, line 10-4,
t3, 4(SI32)
;;
## 17
        c0    stw 20[$r0.4] =
$r0.9   ## bblock 0, line 14-3,
t1, 5(SI32)
;;
## 18
        c0    stw 20[$r0.5] =
$r0.9   ## bblock 0, line 9-5,
t2, 5(SI32)
;;
## 19
        c0    stw 20[$r0.6] =
$r0.9   ## bblock 0, line 10-5,
t3, 5(SI32)
;;
## 20
        c0    stw 24[$r0.4] =
$r0.10   ## bblock 0, line 14-
4, t1, 6(SI32)
;;
## 21
        c0    stw 24[$r0.5] =
$r0.10   ## bblock 0, line 9-6,
t2, 6(SI32)
;;
## 22
        c0    stw 24[$r0.6] =
$r0.10   ## bblock 0, line 10-
6, t3, 6(SI32)
;;
## 23
        c0    stw 28[$r0.4] =
$r0.11   ## bblock 0, line 14-
5, t1, 7(SI32)
;;
## 24
        c0    stw 28[$r0.5] =
$r0.11   ## bblock 0, line 9-7,
t2, 7(SI32)
;;
## 25
        c0    stw 28[$r0.6] =
$r0.11   ## bblock 0, line 10-
7, t3, 7(SI32)
;;
## 26
        c0    stw 32[$r0.4] =
$r0.12   ## bblock 0, line 14-
6, t1, 8(SI32)
;;
## 27
        c0    stw 32[$r0.5] =
$r0.12   ## bblock 0, line 9-8,
t2, 8(SI32)
;;
## 28
        c0    stw 32[$r0.6] =
$r0.12   ## bblock 0, line 10-
8, t3, 8(SI32)
;;
## 29
        c0    stw 36[$r0.4] =
$r0.13   ## bblock 0, line 14-
7, t1, 9(SI32)
;;
## 30
        c0    stw 36[$r0.5] =
$r0.13   ## bblock 0, line 9-9,
t2, 9(SI32)
;;
## 31
```

The above code is for initializing an array. It can be clearly observed that the code generated by VEX compiler has been completely unrolled, which has led to large number of instructions and hence large code size. Also, the multiple functional units have not been properly utilized. There are four functional units in VEX and maximum instructions have single operations. Especially, the memory load operations have not been combined with any other operation. In contrast to this, the code generated by our compiler has not been completely unrolled, leading to reduced code size. Also, all the initial values have not been loaded in the registers as they have been in the assembly code by VEX, but they have been updated in the iterations and same registers have been used. This has led to better utilization of registers and less load/move instructions. As a consequence the compiler packs more operations in an instruction. These improvements have resulted in better ILP. The ILP is majorly affected by the total number of operations and instructions.

The benchmarks were first run on the VEX system and the statistics generated. The commands used to generate the code and then simulate it are:

1) cc –ms-o4 –c <C file>   this command generates the assembly code, the .s file
2) pcntl <assembly file>

Then, the code for the same benchmarks was generated using our compiler, i.e. the assembly code was generated by our compiler. Then, the assembly code was simulated on the VEX system. The command used was:

1) pcntl <assembly file>

The performance statistics for the code generated by VEX compiler and our compiler is shown in the table below:

Benchmarks	**VEX compiler**	**Our compiler**
LL1.c	Operations = 46 Instructions = 25 Reg. moves = 0 Nops = 0 Avg ILP = 1.84	Operations = 49 Instructions = 24 Reg. moves = 0 Nops = 0 Avg ILP = 2.04167
LL5.c	Operations = 44 Instructions = 33 Reg. moves = 0 Nops = 0 Avg ILP = 1.33333	Operations = 51 Instructions = 25 Reg. moves = 0 Nops = 0 Avg ILP = 2.04
LL11.c	Operations = 36 Instructions = 24 Reg. moves = 0 Nops = 0 Avg ILP = 1.5	Operations = 38 Instructions = 17 Reg. moves = 0 Nops = 1 Avg ILP=2.235

LL12.c	Operations = 35 Instructions = 23 Reg. moves = 0 Nops = 0 Avg ILP = 1.52174	Operations = 37 Instructions = 20 Reg. moves = 0 Nops = 0 Avg ILP = 1.85
LL24.c	Operations = 34 Instructions = 13 Reg. moves = 0 Nops = 0 Avg ILP = 2.61538	Operations = 16 Instructions = 5 Reg. moves = 0 Nops = 0 Avg ILP = 3.2

It can be observed from the above table that our compiler has been successful in reducing the number of instructions. This has led to smaller code size.

Another important observation is the Instruction Level Parallelism (ILP). We are aware that VLIW processors make use of pipelining, since they have multiple functional units. The objective in such processors is to pack as many operations in an instruction as possible, so that the functional units can be optimally utilized. ILP is the count of how many operations can be performed in parallel. It can be observed from the above table that our compiler has higher ILP than the code generated by the VEX system.

The above results have been shown graphically in figure 3.

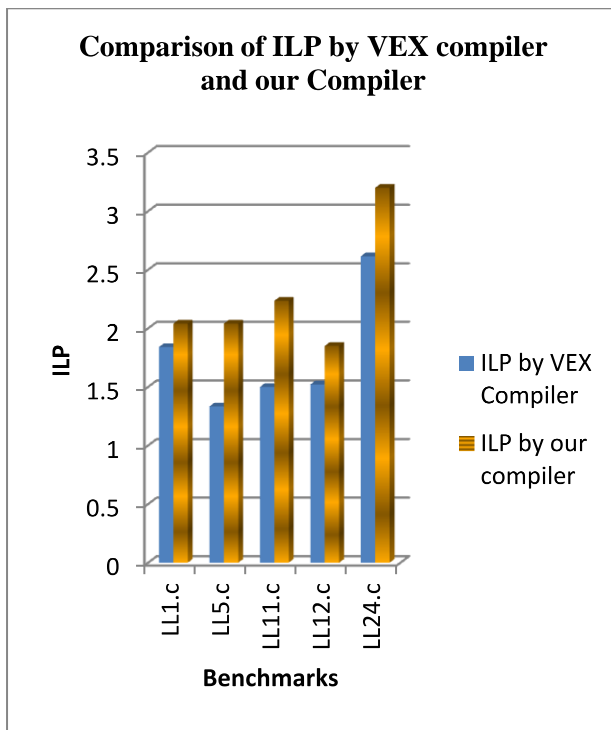

Figure 3: ILP comparison for VEX and our compiler

6. CONCLUSIONS

Exploiting the instruction-level parallelism made available by aggressive superscalar and VLIW processors is one of the hottest topics in the compiler community .The VLIW architecture has many features which enable fast execution of programs, chief among them the use of pipelining. In a more traditional architecture each instruction is fetched, decoded and executed before the next one is fetched, in a pipelined architecture the execution of distinct instructions may overlap one another.

We have developed a retargetable compiler in Visual Basic. It is capable of generating code for a VLIW architecture similar to that of VEX. Our compiler is a user retargetable compiler. The retargetable efforts are intermediate. It has been successfully shown that our compiler has optimally utilized the multiple functional units and has enhanced ILP. Also, the number of instructions is less, which has reduced the code size.

7. REFERENCES

[1] Jain,M.K., Kumar,A., Balakrishnan,M. and Gangwar,A.(2005) Customizing Embedded Processors for Specific Applications, In proceedings of Recent Trends in Practice and Theory of Information Technology, Proc. of NRB Seminar, 10-11 January 2005, NPOL, Cochin, pp. 261-284

[2] Jain, M.K., Balakrishnan, M.and Kumar, A. (2001) ASIP Design Methodologies: Survey and Issues, In proceedings of the Fourteenth International Conference on VLSI Design, 2001, 3-7 Jan. 2001, Pages: 76-81

[3] Jain, M.K., Balakrishnan, M. and Kumar A. (2004), Efficient Technique for Exploring Register File Size in ASIP Design', IEEE TCAD of VLSI, vol. 23, No. 12, pp. 1693-1699, Dec. 2004.

[4] V. Brost, F. Yang, M. Paindavoine and N. Farrugia, "*Multiple Modular VLIW Processors based on FPGA*", Journal of Electronic Imaging SPIE. Vol.16 (2), pp. 023001:1-10, April-June 2007.

[5] J.A. Fisher, P. Faraboschi and C. Young, "*Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*", Elsevier Morgan Kauffman, New York, 2005.

[6] P. Faraboschi, G. Brown and *al.*, "*LX: A technology platform for customizable VLIW embedded processing*", Proc. of the 27th annual International Symposium of Computer Architecture, June 2000.

[7] Hewlett-Packard Laboratories, VEX Toolchain, *http://www.hpl.hp.com/downloads/vex*.

[8] Fisher Joseph A (1991), Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2. *Workshop on Advanced Compilation Techniques for Novel Machine Architecture,* Jerusalem, 1991