Architecture Description Language Driven Design Space Exploration in the Presence of Coprocessors

Prabhat Mishra † Frederic Rousseau ‡ Nikil Dutt † Alex Nicolau †

† Center for Embedded Computer Systems
University of California, Irvine
California 92697 USA
{pmishra, dutt, nicolau}@cecs.uci.edu

‡ System Level Synthesis Group
Laboratoire TIMA
38031 Grenoble cedex, FRANCE
frederic.rousseau@imag.fr

Abstract— Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Shrinking time-to-market, coupled with short product lifetimes create a critical need for rapid exploration and evaluation of candidate System-on-Chip(SOC) architectures. Recent work on language driven Design Space Exploration (DSE) uses Architecture Description Languages (ADL) to capture the processor-memory architecture and automatically generate a software toolkit from the ADL description. We propose in this paper an ADL-based approach to explicitly capture the coprocessor configuration, and perform exploration of the coprocessor architecture along with processor and memory subsystem. We present a set of experiments using the TI C6x architecture to demonstrate the usefulness of our approach.

I. Introduction

Programmable embedded systems are composed of processor, memory subsystem and coprocessors. The coprocessor is used to compute specific functionalities, which the processor is not capable of doing or cannot perform efficiently. During HW/SW co-design of embedded systems, designers need to decide which parts of the functionality is to be implemented using coprocessors. However, to justify the use of a coprocessor the designers need to perform design space exploration. Designers have multiple choices: using a coprocessor to perform a functionality, implement the function in software, add a functional unit in the processor itself to perform the task, modify the existing functional unit(s) to support new operation(s). However, the latter two options are feasible only when modification in the processor core is possible without violating area, timing and power constraints. The first one is the only alternative when processor does not support certain operations e.g, division in TI C6x [12]. In certain architectures e.g., ARM [7], the same operation passes through both processor and coprocessor pipeline; the coprocessor performs the intended task whereas the processor treats the operation as NOP. However, if the operation for the coprocessor is scheduled properly, the processor can execute independent operations while coprocessor is active. Sometimes the processor reads the operands for the coprocessor operation and sends the data to the coprocessor. In this case, the coprocessor behaves as a functional unit in the processor. Usually, coprocessor reads its operands from the memory subsystem. Sometimes it has its own local memory which it uses during computation. The data transfer between coprocessor local memory and main memory is done using DMA controller.

The effect of using coprocessors can be evaluated by performing design space exploration. However, to enable rapid design space exploration there is a need for (i) describing the embedded system (processor, coprocessor, memory subsystem) in higher level of abstraction, and (ii) generating software toolkit (e.g, compiler, simulator, assembler) automatically from the description.

Our approach allows explicit description of coprocessor along with processor and memory subsystem in EXPRESSION ADL [2], permitting co-exploration of the processor, coprocessor and the memory architecture. We generate coprocessor-aware software toolkit that allows rapid design space exploration.

This paper is organized as follows. Section II presents related work addressing ADL-driven DSE approaches. Section III outlines our approach and the overall flow of our environment. Section IV presents the coprocessor description in EXPRESSION ADL [2]. Section V describes how we generate coprocessor-aware software toolkit. Section VI presents design space exploration experiments using the software toolkit. Section VII concludes the paper.

II. Related Work

Recent approaches on language-driven design space exploration ([2], [3], [5], [13], [14]), use ADLs to capture the processor architecture, automatically generate a software toolkit for that processor, and provide feedback to the designer on the quality of the architecture.

nML [3] has been used by the retargetable code generation environment to describe DSP and ASIP processors. In ISDL [5], constraints on parallelism are explicitly specified through illegal operation groupings. This could be tedious for complex architectures like DSPs which permit operation parallelism (e.g. Motorola 56K) and VLIW machines with distributed register files (e.g. TI C6X). MDes [13] allows only a restricted retargetability of the simulator to the HPL-PD processor family. MDes permits the description of the memory system, but is limited to the traditional cache hierarchy. LISA [14] captures VLIW DSP and RISC architectures. An ADL-based approach for processor-memory co-exploration is presented in [1].

While these approaches explicitly capture the processor and memory features to varying degrees, to our knowledge, no previous approach allows explicit capture of a coprocessor along with processor and memory specification, and the attendant tasks of generating a software toolkit that fully exploits the coprocessor features.

III. Our Approach

Figure 1 shows the flow in our approach. In our IP library based Design Space Exploration (DSE) scenario, the designer starts by selecting a set of components from a processor IP library, memory IP library, and coprocessor IP library. The ADL description (containing a mix of such IP components and custom blocks) is then used to generate the information necessary to target both compiler and simulator to the specific architecture.

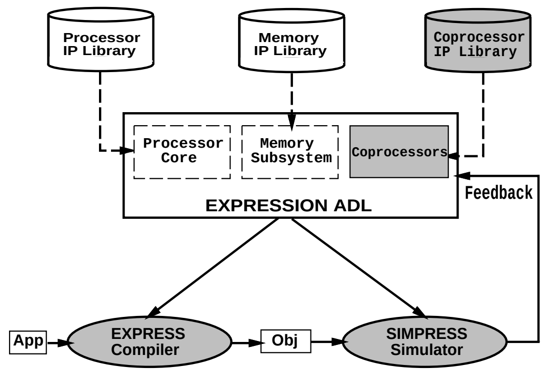

Figure 1: Flow in our approach

We capture coprocessor description in EXPRESSION ADL [2] along with processor and memory description. We generate from this ADL description the EXPRESS compiler [6] and SIMPRESS simulator [8] that can exploit the features available in the coprocessor, allowing for detailed feedback on the coprocessor architecture and its match to the target applications.

IV. Coprocessor description in EXPRESSION

In order to explicitly describe any coprocessor in ADL, we need to capture both the structure and behavior of the coprocessor. In this section we present how to describe coprocessors in EXPRESSION ADL by way of illustrative examples. Figure 2 shows a simplified model of the example TI C6x architecture with a sample coprocessor. The pipeline paths are shown using solid lines whereas the data transfer paths are shown using dotted lines. The TI C6211 is an 8-way VLIW DSP processor, composed of 4 fetch stages (PG, PS, PR, PW), 2 decode stages (DP, DC), followed by 8 functional units (L1, S1, M1, D1, L2, S2, M2, D2).

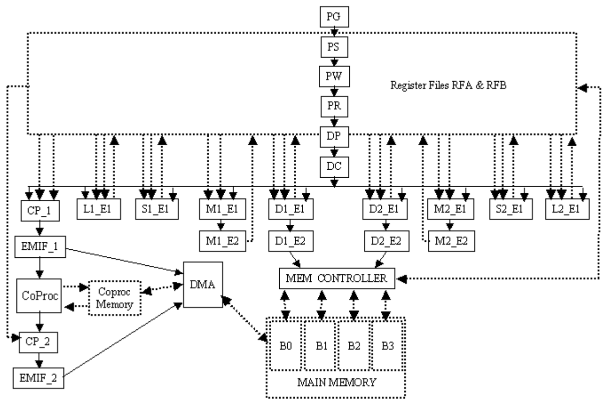

Figure 2: TI C62x Architecture with Coprocessor

To describe the structure of the coprocessor we specify each pipeline stage of the coprocessor along with their characteristics (e.g, timing, parallelism etc.). The pipeline paths are described in a hierarchical manner. Figure 2 has seven pipeline stages: four fetch stages (PG, PS, PR, and PW), followed by two decode stages (DP and DC), and finally an execute stage (Execute). The ADL description of the pipeline path is shown below.

```
(PIPELINE PG PS PW PR DP DC Execute)
```

The execute stage has nine parallel (ALTERNATE) pipeline paths: one coprocessor pipeline, two L pipelines (L1, L2), two M pipelines(M1, M2), two S pipelines (S1, S2), and two D pipelines (D1, D2). The ADL description of the *Execute* stage is shown below.

```
(Execute (ALTERNATE COPRO L1 S1 M1 D1 D2 M2 S2 L2))
```

Each of these pipelines again has its own pipeline description. For example, coprocessor pipeline (COPRO) has five pipeline stages: CP_1, EMIF_1, CoProc, CP_2, and EMIF_2.

```
(COPRO (PIPELINE CP_1 EMIF_1 CoProc CP_2 EMIF_2))
```

The coprocessor instruction is decoded in CP_1 stage to determine the size of input data and starting address in the main memory. The EMIF_1 stage requests the DMA to transfer the data from the main memory to the coprocessor memory, using efficient access modes if needed. The CoProc stage performs the intended computation (e.g., vector multiply, FFT etc.) using the coprocessor memory for accessing input operands. Results are stored back in the coprocessor memory. The CP_2 stage examines the size of the result and starting address in the main memory to store the results. Finally, EMIF_2 requests the DMA to transfer the data from coprocessor memory to main memory. The example below shows the characteristics viz., timing, opcodes supported, parallelism (CAPACITY) etc. for the *CoProc* unit.

```
(CoproUnit CoProc
    (CAPACITY 1)
    (TIMING (all 4))
    (OPCODES COPRO_instr)
    (LATCHES .....)
    (PORTS ....)
)
```

The complete description of the coprocessor that includes five coprocessor stages, DMA controller, coprocessor memory (in STORAGE section), pipeline latches, ports, connections etc. can be found in [10].

Behavior of the coprocessor is captured in terms of the operations it supports. For example, the EXPRESSION description for the vector multiplication operation is shown below.

```
(OP_GROUP COPRO_instr
    (OPCODE VectMul
        (OPERANDS (_SOURCE_1_ mem) (_SOURCE_2_ mem)
                  (_DEST_ mem) (_LENGTH_ immediate)
    )
)
```

Unlike normal instructions whose source and destination operands are register type (except load/store), here source and destination operands are memory type. The _SOURCE_1_ and _SOURCE_2_ fields refer to the starting addresses of two source operands for the multiplication. Similarly _DEST_ refers to the starting address of destination operand for the multiplication. The _LENGTH_ field refers to the vector length of the operation that has immediate data type.

The explicit representation of the coprocessor structure and behavior allows the compiler to exploit the organization of the coprocessor during operation scheduling, and the simulator to provide detailed feedback on the internal coprocessor traffic.

The pipelining and parallelism between the coprocessor operations are described in EXPRESSION through pipeline and data-transfer paths. Pipeline paths represent the ordering between pipeline stages in the architecture (represented as solid lines in Figure 2). For example, a coprocessor operation traverses first 4 fetch stages of the processor, followed by the 2 decode stages, and then it goes through 5 coprocessor stages. It also traverses the data transfer paths for reading operands and writing results (represented as dotted lines in in Figure 2). For example, a coprocessor read operation traverses *CoProc* followed by *CoProc-Memory* which gets the data from main memory using DMA. Thus the pipeline path traversed by the example coprocessor operation is:

```
(PIPELINE PG, PS, PR, PW, DP, DC, CP_1,
          EMIF_1, CoProc, CP_2, EMIF_2)
```

In this manner EXPRESSION can model a variety of coprocessor modules and their characteristics. The EXPRESSION description can be used to drive the generation of both coprocessor-aware compiler and cycle-accurate structural coprocessor simulator, as described in Section V.

V. Software Toolkit Generation

In this section we briefly outline how we generate retargetable compiler and simulator which exploit the features of the coprocessor.

We are able to generate co-processor aware retargetable simulator due to the use of functional abstraction based primitives. We define each stage of the coprocessor unit using parameterized functions. Each function is further composed of generic sub-functions which allows a finer granularity of architectural exploration. For example, the generic *CoProc* unit uses three sub-functions (e.g., *ReadOperand*, *ComputeResult*, and *WriteOperand*) as shown below. The parameters for the generic functions (e.g., src1Addr, length etc.) are obtained from the EXPRESSION description of the coprocessor as presented in Section IV. The *ComputeResult* sub-function uses pointer to the operation to be performed (*funcPtr*) as a parameter. For example, the *funcPtr* may point to a function that performs vector multiplication.

```
CoProc (src1Addr, src2Addr, destAddr, length, funcPtr )
{
    // ReadSrc1, the start address of Src1 vector and length
    // of the vector are needed to retrieve the values.

    S1 = ReadOperand(src1Addr, length);

    // ReadSrc2
    S2 = ReadOperand(src2Addr, length);

    // Perform the operation
    D0 = ComputeResult(S1, S2, funcPtr);

    // Write the result back.
    WriteOperand(destAddr, length);
}
```

Similarly, we use the generic model of coprocessor memory and DMA controller. The detailed description of all the generic abstractions used to retarget the simulator are too long to describe in this paper, and can be found in [9].

The coprocessor related information available in the EXPRESSION description are used to retarget the compiler as well. For example, to schedule the operation, the instruction description, and the pipeline and timing information of the coprocessor are used to generate reservation tables [4]. Considering the fact that DMA can take a long time to access (exact timing is known) the main memory, the compiler can schedule operations using reservation tables such that when the coprocessor is busy, the functional units of the processor can execute independent operations. To obtain optimized code of the application the trailblazing percolation scheduling [11] is used.

In this manner we can generate both coprocessor-aware compiler and cycle-accurate structural coprocessor simulator, and thus enable design space exploration and co-design of the coprocessor and processor-memory architecture. For more details on the coprocessor description in EXPRESSION and automatic software toolkit generation, please refer to [10].

VI. Experiments

We present here a set of experiments to show the usefulness of our approach. Our goal is to study the performance impact of using coprocessor to support vector multiplication.

A. Experimental Setup

We used a set of benchmarks from DSPStone fixed point benchmarks that uses vector multiplication. We have chosen TI C62x architecture for the exploration. The M unit of the TI C62x can be used for vector multiplication. The M unit executes the multiplication operation (MUL) iteratively as shown below.

```
// Vector Multiplication code in Application Program
for (i=0; i < n; i++)
    z[i] = a[i] * b[i];

// Translated pseudo assembly that can run on TI C62x
    MOV i, 0
L5: LOAD x, mem_address(a[i])
    LOAD y, mem_address(b[i])
    MUL  t, x, y
    STORE t, mem_address(z[i])
    INC i
    LT  $cc i n)
    IF $cc L5

// Equivalent instruction for the coprocessor
VectMul(a, b, z, n);
```

However, the same vector multiplication can be performed using a coprocessor. The equivalent coprocessor instruction is shown above. Naturally, the architectural configuration that uses coprocessor is costlier. In both cases (with or without coprocessor) we described the architecture using EXPRESSION description and generated EXPRESS compiler and SIMPRESS simulator.

B. Results

Figure 3 presents a subset of experiments we ran, showing the total cycle counts for the set of benchmarks for two different architectural configurations, viz., with coprocessor and without coprocessor. The light bar presents the number of execution cycles when the functional unit is used for the multiplication whereas the dark bar presents the number of execution cycles when the co-processor is used. The configuration with coprocessor shows 29% improvement for *dotproduct*, vector multiplication dominated DSP kernel whereas it shows only 5% improvement for *fir2dim* since vector multiplication is only a minor part of the program. The performance improvement is due to the fact that coprocessor uses its local memory and rely on efficient DMA transfer. Moreover, functional units (e.g., M1) operate in register-to-register mode whereas co-processor operates on its memory-memory mode. As a result the register pressure, and therefore spilling, is reduced in the presence of coprocessors.

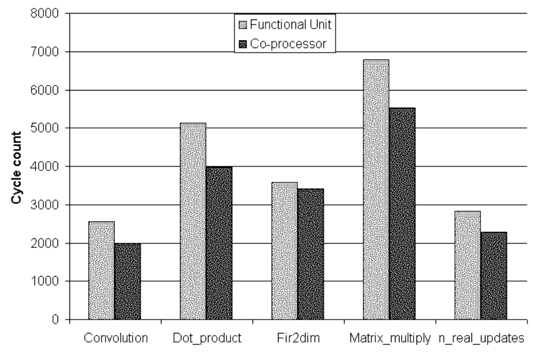

Figure 3: Performance with or without co-processor

We ran another set of experiments for the benchmark *convolution* with vector multiplication of different vector lengths. Table I presents the results of this experiment. The first column represents the length of vectors in vector multiplication operation in the *convolution* benchmark. The second column represents the number of cycles needed to execute the program when functional unit is used for vector multiplication. The third column presents the number of cycles needed to execute the program when co-processor is used for performing vector multiplication instead of the functional unit. The last column shows performance improvement for using the coprocessor. For smaller vector lengths, functional unit performs better since co-processor needs set up cycles. As expected, the coprocessor performs better when vector length is large. The use of coprocessor can deliver upto 29% performance improvement for the *convolution* benchmark.

Thus, using our coprocessor-aware ADL-based design space exploration approach, we obtained design points with varying cost and performance. Note that this cannot be determined through analysis alone; the cus-

Table I: Performance Analysis for *convolution* benchmark with co-processor for varied vector size

Vector length	Functional Unit (cycle count)	Coprocessor (cycle count)	% improve
1	45	47	-4.26
2	82	74	10.81
4	162	135	20.00
8	322	257	25.29
16	642	504	27.38
32	1282	1000	28.20
64	2562	1992	28.61
128	5122	3976	28.82
256	10242	7944	28.93
512	20482	15880	28.98
1024	40962	31752	29.01

tomized coprocessor must be explicitly captured, and the applications have to be executed on the configured architecture, as we demonstrated in this section.

VII. Summary

This paper proposed an ADL driven design space exploration methodology in the presence of coprocessors. We capture the coprocessor description in EXPRESSION along with processor and memory description and generate software toolkit that exploits the coprocessor features. We demonstrated the power of our approach by performing co-exploration of TI C62x processor, coprocessor, and memory architecture.

Our ongoing work targets the study of cost measures for the local memory, DMA and coprocessor to decide the best cost/performance figure. We plan to perform exploration using larger examples, to study the impact of applications on the coprocessor and overall performance, as well as on system power.

Acknowledgments

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Corporation. We would like to acknowledge Ashok Halambi and Peter Grun for their contribution to the coprocessor modeling work.

References

[1] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Intl. Conf. on VLSI Design 2001*, Bangalore, India, 2001.

[2] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[3] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[4] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, San Jose, CA, 1999.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. *DAC*, 1997.

[6] A. Halambi, N. Dutt, and A. Nicolau. Customizing software toolkits for embedded systems-on-chip. In *DIPES 2000*, 2000.

[7] http://www.arm.com/. *ARM7TDMI-S.*

[8] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.

[9] P. Mishra, J. Astrom, N. Dutt, and A. Nicolau. Functional abstraction of programmable embedded systems. Technical Report UCI-ICS 01-04, University of California, Irvine, 2001.

[10] P. Mishra, F. Rousseau, N. Dutt, and A. Nicolau. Coprocessor codesign for programmable architectures. Technical Report UCI-ICS 01-13, University of California, Irvine, 2001.

[11] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. *ICPP*, 1993.

[12] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*, 1998.

[13] http://www.trimaran.org. *The MDES User Manual*, 1997.

[14] V. Zivojnovic, S. Pees, and H. Meyr. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.