Proceedings of 1st Workshop on Resource Awareness and Application Autotuning in Adaptive and Heterogeneous Computing (RES4ANT) 2016

Design Space Exploration of Flexible Heterogeneous Dual-Core Processor using MIPS and Extended OISC: A Case Study

Noriaki Sakamoto† Tanvir Ahmed† Jason H. Anderson‡ Yuko Hara-Azumi†∗
†Tokyo Institute of Technology ‡University of Toronto ∗JST PRESTO
Email: †{noriakis, tanvira, hara}@cad.ce.titech.ac.jp ‡janders@ece.utoronto.ca

Due to the growth of Internet-of-Things (IoT) technology, the diversity of embedded systems applications is rapidly increasing nowadays. Although homogeneous multi-core processors have been commonly used to achieve good performance in a variety of commercial products, it is difficult to satisfy contradicting design constraints (i.e., low area (cost), low power/energy and good performance) by such common architecture platforms for a variety of IoT applications because they have more stringent constraints. Namely, there is no one-size-fits-all solution and thus application-specific heterogeneous processors become more and more important.

The design decision on heterogeneous multi-core processors is very difficult due to quite large design space—*how many cores should be used?, what functionality (instructions) should those cores have?, what instructions set should they have?*, etc. One approach is to start with a general processor (VLIW) and trim resources and instructions which are not (frequently) used by the target applications [3]. However, such trimmed processors lack versatility and cannot perform other applications. Versatility is crucial for IoT devices because they are likely to have the need of multi-tasking and/or future software update.

One useful solution is, in case of dual-core processors, to start with two very different processors; one is rich and the other is poor in resources. Depending on the target applications, they can be both customized efficiently. Such pioneering-yet-elementary works have been presented in [4], [1]. These works both use MIPS as the rich processor and Subleq [2] as the poor processor—Subleq is one of One-Instruction-Set Computers (OISC) which have a single type of instruction only but are Turing-Complete. The subleq instruction is given with three operands, `subleq A B C`, and performs subtraction and less-than-equal in one instruction as follows:

$$\begin{aligned} r &\leftarrow \mathrm{mem}[B] - \mathrm{mem}[A] \\ \mathrm{mem}[B] &\leftarrow r \\ PC &\leftarrow \begin{cases} C & \text{if } r \leq 0 \\ PC + 3 & \text{if } r > 0 \end{cases} \\ &\text{Halt if } C < 0 \end{aligned}$$

The work [4] considers the trimming of MIPS while preserving the versatility as compensating the trimmed functionality by Subleq, enabling exploration of trade-offs in area, performance, and power depending on the application. On the other hand, the work [1] adds functionality (instructions) in Subleq to improve its efficiency—Subleq takes many instructions to perform the equivalent MIPS instruction (e.g., five subleq instructions to perform an addition operation). What remain to be solved in these works are (1) to mitigate performance degradation when using Subleq and (2) to enable bidirectional customization on both MIPS and Subleq. More specifically, for the issue (1), performance degradation is extremely large for arithmetic operations, especially for *multiplications and right shift operations handling small values*[1] which often appear in media and IoT applications; and for the issue (2), the works [4], [1] consider customization of only either of MIPS and Subleq (onedirectional customization). Extending Subleq for better performance with minimum area overhead and integrating the extended Subleq in the framework [4] will help the designers explore better design space efficiently.

In this work, we first propose an extended Subleq, **Subleq**$_\ominus$, with one additional instruction which can fully exploit the resources of the original Subleq. A straightforward approach of Subleq extension is to add an instruction which can speedup particularly inefficient operation, such as multiplication. This approach can speedup only such specific operations not but other types of operations. Moreover, the area overhead will become very large, spoiling the merit of the simplicity and light-weightness of Subleq. Contrary to that, we break down the cause of inefficiency—a number of instructions for multiplication and right shift with small values—and resolve the root cause by adding a bit-reversal subleq instruction, subleq$_\ominus$, which first bit-reverses two operands, performs subleq for these two bit-reversed values, bit-reverses back the result, and then performs less-than-equal. This extension efficiently resolves the aforementioned inefficiency of the original Subleq. Furthermore, another merit is that most resources of Subleq can be exploited and the additional circuit is only for pre- and post-processing bit-reversal, mitigating the area overhead. The

[1]For these operations, Subleq needs to find the MSB with '1' which takes a number of instructions.

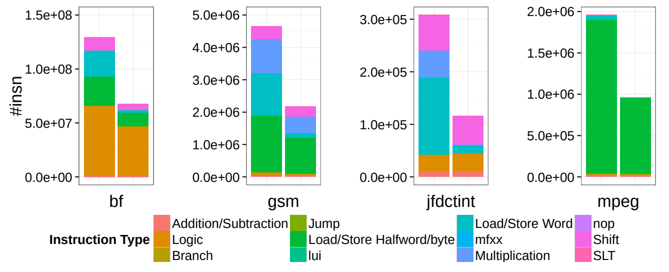

Fig. 1. Comparison of instruction counts.

TABLE I Synthesized area and frequency.

	Host Only	Host w/ Co-Processor		
		M	**MS+**	**ALL**
LEs	1821	1600	1232	1141
Ratio	1.00	0.88	0.68	0.63
		M$_\ominus$	**MS+**$_\ominus$	**ALL**$_\ominus$
LEs		1674	1306	1215
Ratio		0.92	0.72	0.67
# of Multipliers	6	0	0	0
Frequency (MHz)	150.7	154.6	169.9	176.8
Ratio	1.00	1.03	1.13	1.17

extended ISA, `subleq`$_\ominus$ `A B C`, is defined as follows:

$$r \leftarrow \begin{cases} \text{mem}[B] - \text{mem}[A] & \text{if } C > 0 \\ \ominus(\ominus\text{mem}[B] - \ominus\text{mem}[A]) & \text{else if } C < 0 \end{cases}$$

$$\text{mem}[B] \leftarrow r$$

$$PC \leftarrow \begin{cases} |C| & \text{if } C > 0 \text{ and } r \leq 0 \text{ or} \\ & \text{if } C < 0 \text{ and } \ominus r \leq 0 \\ PC + 3 & \text{otherwise} \end{cases}$$

$$\text{Halt if } C = 0$$

As defined above, Subleq$_\ominus$ dynamically selects the instruction to be used based on the values—subleq for large values and subleq$_\ominus$ for small values—so that better performance is achieved. The dynamic selection takes overhead (on average five instructions) which is negligibly small compared with the speedup effect.

Performance improvement by this extension (i.e., Subleq$_\ominus$ over Subleq, both used as a single-core processor) is evaluated for four practical applications, `bf`, `gsm`, `jfdctint`, and `mpeg`, which have different features in the ratio of instruction types. As seen from Fig. 1, independent of the major types of instructions, Subleq$_\ominus$ achieves significant reduction in dynamic instructions, i.e., performance improvement.

Next, in this work as a case study, we evaluated our Subleq$_\ominus$ in the heterogeneous dual-core processor similarly as in [4], where the trade-offs are realized by varying the functionality (instructions) of the host processor (MIPS)—we will see how the trade-offs would improve by replacing Subleq with our Subleq$_\ominus$ for FPGA Cyclone II. Table I tabulates the circuit area (the number of logic elements (LEs) and built-in multipliers) and clock frequency of different heterogeneous processors against a baseline single-core MIPS. For heterogeneous processors, **M**, **MS+**, and **ALL** indicate the functionality (multiplication, arithmetic and shift, and all, respectively) removed from the host processor and alternatively performed by the co-processor. Attached without and with $_\ominus$mean that the co-processor is Subleq and our Subleq$_\ominus$, respectively. The table shows that as expected, our Subleq$_\ominus$ has small area overhead (4.6%–6.4%) and no effect on clock frequency. Then in Fig. 2, we compared the design space of circuit area (x-axis) and execution time (y-axis) of these heterogeneous processors for four practical benchmarks. In most cases especially bigger applications, our Subleq$_\ominus$ enables to explore better design space, demonstrating the effectiveness.

In summary, our key contribution is to enable bidirectional customization of both resource-rich and resource-poor processors in a heterogeneous dual-core processor—the extension of the resource-poor co-processor (Subleq) by fully exploiting its original structure and resolving the root cause of inefficiency and the trimming of too rich functionality at the resource-rich host processor (MIPS). Although we followed the structure of [4], [1] (MIPS-Subleq) in this work, other types of processors can be also used for both host and co-processors. Also, our Subleq$_\ominus$ has good applicability from as a single-core processor to as many-core processors [2]. Our future work is to consider application-specific extension of Subleq to further enhance the design space.

References

[1] S. Ananthanarayan, S. Garg, and H.D. Patel, “Low cost permanent fault detection using ultra-reduced instruction set co-processors,” Design, Automation & Test in Europe, pp.933–938, Mar. 2013.

[2] O. Mazonka and A. Kolodin, “A simple multi-processor computer based on subleq,” May 2011.

[3] J. Matai et al., “Trimmed VLIW: Moving application specific processors towards high level synthesis ,” Electronic System Level Synthesis Conference, pp. 11–16, Jun. 2012.

[4] T. Ahmed et al.,“Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC,” Embedded and Ubiquitous Computing, pp. 114–123, Oct. 2015.

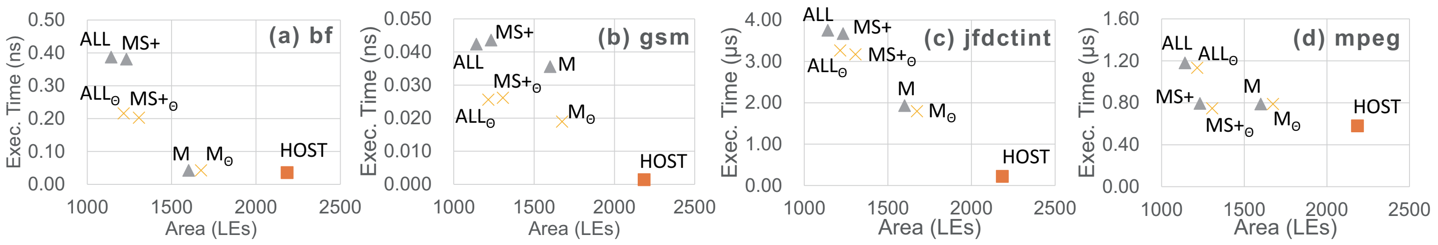

Fig. 2. Area vs. execution time.