Montgomery Modular Multiplication Algorithm for Multi-Core Systems

Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede

Katholieke Universiteit Leuven,ESAT/SCD-COSIC,
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
{Junfeng.Fan,Kazuo.Sakiyama,Ingrid.Verbauwhede}@esat.kuleuven.be

Abstract. This paper presents an efficient software implementation of the Montgomery modular multiplication algorithm on a multi-core system. A prototype of general multi-core systems is designed with GEZEL. We propose a new instruction scheduling method for multi-core systems that can reduce the number of data transfers between different cores. Compared to the implementations on a single-core system, the performance can be improved by a factor of 1.87 and 3.68 when 256-bit modular multiplication being performed on a 2-core and 4-core system, respectively.

Key words: Montgomery Modular Multiplication, Multi-core, Parallel Computation

1 Introduction

Modular multiplication is a fundamental operation in many popular Public Key Cryptography (PKC) algorithms such as RSA [1] and ECC [2, 3]. As the division operation in modular reduction is time-consuming, Montgomery [4] proposed a new algorithm where division is avoided. An integer Z is represented as $Z \cdot R \mod M$, where M is the modulo and $R = 2^r$ is a radix that is coprime to M. This representation is called Montgomery residue. Multiplication is performed in this residue, and division by M is replaced with division by R. This algorithm can be easily implemented on general purpose processors. However, due to the highly intensive computation, software implementations are offen not fast enough. Many hardware implementations [5–7] were proposed to improve the performance.

The increasing use of multi-core systems have opened another window for improving the performance of software implementations. Processor vendors have published various dual-core [8] and quad-core [9] processors for personal computers. Even for embedded systems several multi-core processors [10, 11] are now available. Therefore, multiple cores in the system can be utilized to perform the intensive computation and the software implementations of the Montgomery algorithm can then be accelerated by parallel computation.

When performing parallel computation, task scheduling is highly dependent on the hardware architecture. If the architecture is based on a super-scalar processor, the task will be automatically partitioned. In this paper, we consider

general multi-core systems that do not have this feature. We use a Very Long Instruction Word (VLIW) processor as a prototype. This processor can be configured to have 1, 2, 4, 8 or even more cores. Each core can work separately. To be general, only the very basic instructions are supported. The Montgomery algorithm is partitioned in algorithm level and tasks are mapped to each core. We explore two different scheduling methods to find bottlenecks.

The rest of the paper is organized as follow. Section 2 briefly reviews previous work on the Montgomery algorithm and its parallel implementations. In section 3, we describe the multi-core architecture of our platform. Two instruction scheduling methods are proposed in section 4 and comparison between them is given in section 5. Finally, we show implementation results in section 6 and conclude the paper including future work in section 7.

2 Previous Work

The Montgomery modular multiplication algorithm was designed to avoid division in modular multiplications. Given two n-bit inputs, X and Y, this algorithm gives $Z = X \cdot Y \cdot R^{-1} \bmod M$, where R equals to 2^n and M is the n-bit modulo. A modified Montgomery multiplication algorithm was proposed to avoid the conditional final substraction by choosing a suitable R [12]. Algorithm 1 shows the Montgomery algorithm with the conditional substraction.

Algorithm 1 Radix-2^w Montgomery modular multiplication (FIOS) [13]

Input: integers $M = (M_{s-1}, ..., M_0)_r$, $X = (X_{s-1}, ..., X_0)_r$, $Y = (Y_{s-1}, ..., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$, $s = \lceil \frac{n}{w} \rceil$, $R = r^s$ with $gcd(M, r) = 1$ and $M' = -M^{-1} \bmod r$.
Output: $X \cdot Y \cdot R^{-1} \bmod M$

1: $Z = (Z_{s-1}, ..., Z_0)_r \leftarrow 0$
2: **for** $i = 0$ to $s - 1$ **do**
3: $\quad T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M' \bmod r$
4: $\quad Z \leftarrow (Z + X \cdot Y_i + M \cdot T)/r$
5: **end for**
6: **if** $Z > M$ **then**
7: $\quad Z \leftarrow Z - M$
8: **end if**
9: return Z

As shown in Algorithm 1, the operands X, Y and M are divided into w-bit words. In the beginning of each iteration, $X_0 \cdot Y_i$ is calculated to generate T. After the generation of T, the multiplication of $X \cdot Y_i$ and reduction of C are performed together by computing $Z = Z + X \cdot Y_i + M \cdot T$. After that, Z_0 always becomes 0. The division of Z by r is performed by shifting Z one word to the right. After s iterations and one conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is obtained. As Algorithm 1 scans the operands X and M from Least Significant Bit (LSB) to Most Significant Bit (MSB) simultaneously, it is also called Finely Integrated

Operand Scanning (FIOS). It is possible to perform $Z = Z + X \cdot Y_i$ first, and then $Z = Z + M \cdot T$. This modified algorithm is called Coarsely Integrated Operand Scanning (CIOS) [13] and is presented in Algorithm 2.

Algorithm 2 Radix-2^w Montgomery modular multiplication (CIOS) [13]

Input: integers $M = (M_{s-1}, ..., M_0)_r$, $X = (X_{s-1}, ..., X_0)_r$, $Y = (Y_{s-1}, ..., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$,$s = \lceil \frac{n}{w} \rceil$,$R = r^s$ with $gcd(M, r) = 1$ and $M^{'} = -M^{-1} \bmod r$.
Output: $X \cdot Y \cdot R^{-1} \bmod M$

```
1: Z = (Z_{s-1}, ..., Z_0)_r ← 0
2: for i = 0 to s − 1 do
3:     T ← (Z_0 + X_0 · Y_i) · M' mod r
4:     Z ← (Z + X · Y_i)
5:     Z ← (Z + M · T)/r
6: end for
7: if Z > M then
8:     Z ← Z − M
9: end if
10: return Z
```

So far, many task scheduling methods have been proposed. Kaihara *et al.* [14] designed a bipartite multiplier, where the modular multiplication was divided into two separated tasks. Besides, systolic array [15] is deployed in hardware implementations. Various implementations [16–18] of systolic array were proposed to improve the performance. However, most of the scheduling methods are targeting a fast hardware implementation, data transfers between PEs are almost free since they can be performed with a hardwired communication. In general purpose multi-core systems, different cores exchange data via shared memory. Thus, frequent data transfers can make a heavy overhead. Therefore, those scheduling methods need to be modified to fit software implementations.

In this paper we propose a new scheduling method, which can efficiently reduce the data transfers between different cores. This method is highly scalable and can achieve high performance.

3 Our Design Platform

It doesn't make sense to fix the hardware architecture (i.e. the number of cores) and explore the best software algorithm for the fixed hardware configuration. The hardware/software co-design with a multi-core system, the main focus of this paper, needs an environment to get a quick and correct evaluation of cost and performance for various hardware configurations and software programs. Thus, we use a simulation environment, called GEZEL [19], which allows us to estimate immediate system performance in a cycle-accurate manner before synthesizing the entire design.

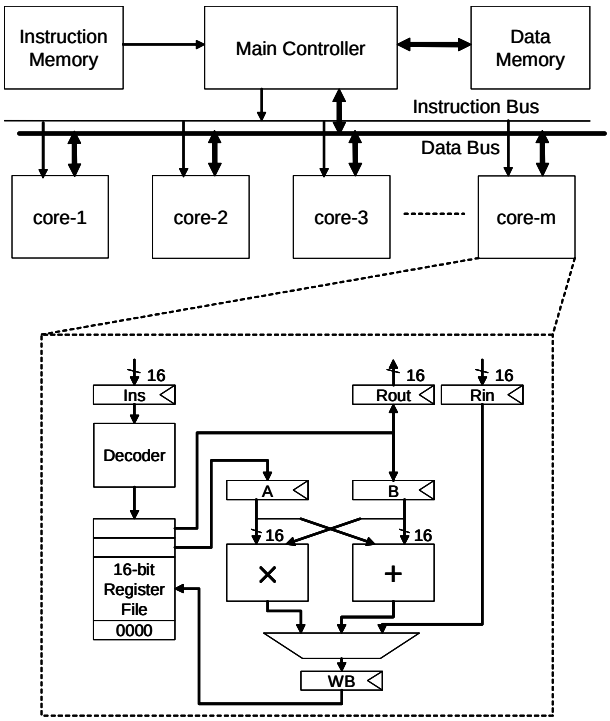

Fig. 1. Platform architecture. ($w = 16$).

General multi-core systems can have various architectures and corresponding memory organizations. For instance, they may contain symmetrical cores with a shared memory, or a master CPU with several slave CPUs/DSPs connected to the system bus. To be as general as possible, we use a VLIW architecture processor. The purpose of this prototype processor is to explore different algorithms on multi-core systems.

As shown in Figure 1, this platform consists of a main controller, a data memory, an instruction memory and several cores. Only the main controller can access the instruction memory and the data memory. The main controller fetches instructions from the instruction memory and dispatches them to all cores in parallel via the instruction bus. Each core executes arithmetic instructions in parallel, and stores the results in its register file. The data memory has only one read/write port, therefore, a single data memory access is allowed in each cycle.

We denote w as the operation size of w-bit core. A 16-bit ($w = 16$) core is also shown in Figure 1. It is a highly simplified Load/Store CPU. It has a instruction decoder, a register file with 16 general 16-bit registers and one status register. The Arithmetic Logic Unit(ALU) includes one 16-bit multiplier and one 16-bit adder. It also has an output register to store the data that will be written to the data memory, and an input register to buffer the data from the data memory.

Table 1. Instruction sets for one core.

Opcode 4-bit	**Operand 1** 4-bit	**Operand 2** 4-bit	**Operand 3** 4-bit	**Description**
`Nop`				No operation
`Load`	`Ri`	`#Addr`		Load the data from location `Addr` of the data memory into register `Ri`
`Store`	`Ri`	`#Addr`		Store the data of register `Ri` to location `Addr` or the data memory
`Mul`	`Ri`	`Rj`	`Rk`	`{R(i+1),Ri} = Rj· Rk`
`Add`	`Ri`	`Rj`	`Rk`	`{Ca,Ri} = Rj + Rk`, `Ca` is the carry out and is stored in the status register
`Adc`	`Ri`	`Rj`	`Rk`	`{Ca,Ri} = Rj + Rk + Ca`
`Sub`	`Ri`	`Rj`	`Rk`	`Ri = Rj - Rk`

Both of them are 16-bit. One 32-bit Write Back (WB) register is also used to store data from the ALU. The bit-length of both data-path and registers are doubled if it is configured as a 32-bit ($w = 32$) core.

The cores here support a simple Load/Store Instruction Set Architecture (ISA). As shown in Table 1, this simplified ISA has only 7 general instructions. Here `#Addr` denotes memory address. Instructions for each core are of 16-bit long. All the arithmetic operations are performed among data stored in the local register file. When data needs to be moved from one core to another, it is first stored to the data memory, then it is loaded by the destination core. Cores in this platform support a 4-stage instruction pipelining.

4 Instruction Scheduling

Before we schedule the instructions, the data dependency is analyzed. The main dependency in the Montgomery algorithm is due to the carries of additions. Taking FIOS shown in Algorithm 1 as an example, in each iteration, Z_j is replaced by $(Z_j + (X \cdot Y_i)_j + (M \cdot T)_j + Ca)$, where Ca is the carry. The data dependency in one iteration is shown in Figure 2. Obviously, $X_j \cdot Y_i$, for any $0 \leq i, j \leq s-1$, is only dependent on the operands X and Y. We can also calculate $M_j \cdot T$ immediately after the generation of T. The products with the same weight of Z_j and the carry from Z_{j-1} are accumulated to Z_j, generating a new Z_j and 2-bit carries. As a result, Z_j can only be generated after carry from Z_{j-1} is ready.

As shown in Figure 2, we need to add Z_j with four w-bit data and 2-bit carries. In hardware implementations, cascaded Carry Save Adders (CSAs) can be used to construct a 6-to-2 CSA. The carry can also be saved in a 2-bit register or transferred to another PE. However, in general purpose processors these special features are not available. Normally only general adders with a

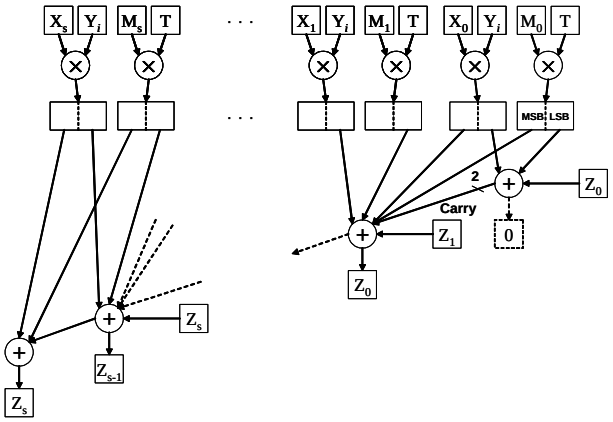

Fig. 2. Data dependency of FIOS Montgomery algorithm.

fixed length are used. The carry is saved in the status register after an `Add` instruction. In order to keep the 1-bit carry for future use, one instruction is needed to copy it from the status register to a general register. It will be very inefficient to use carries generated by another core, since it needs to be stored to register file first, and then transferred via the data memory.

Therefore, it will be desirable to partition the algorithm so that carry is only used in the core where it was generated. In [6], Tenca and Koç proposed an iteration-based scheduling method. In this method each Processing Element (PE) performs one iteration of the loop in Algorithm 1. This method is attractive because carries are only used in the local PE. Note that this method was originally designed for a hardware implementation. Here we map this algorithm to general purpose multi-core systems. Figure 3 shows the scheduling method, denoted as method-I, for 256-bit Montgomery multiplication for a 4-core system. As $n = 256$ and $w = 16$, sixteen iterations are needed. Core-1 performs the first iteration and generates Z_0 to Z_{15} one bye one. Each word is transferred to core-2 as soon as it is generated. Core-2 then performs the second iteration and then transfers Z_0 to Z_{15} to core-3. After 4 iterations $Z = (Z_{15}, ..., Z_0)$ is transferred back to core-1 from core-4 and the 5^{th} iteration begins. As in total 16 iterations are required, each core needs to perform 4 iterations. After a conditional substraction, the result is obtained.

Though the method-I can avoid carry transfers between cores, transferring $(Z_{s-1}...Z_0)$ causes a heavy overhead. In Figure 3 the transfers of $(Z_{s-1}...Z_0)$ are denoted as arrows. For each iteration $s = \lceil \frac{n}{w} \rceil$ arrows are required to transfer Z. Since one modular multiplication contains s iterations, $s(s-1)$ arrows are needed during the whole loop. Let N_{arrow} be the number of arrows, then N_{arrow} is $s(s-1)$. In Figure 3 we have $s = 16$, therefore $N_{arrow} = 240$.

Note that in order to generate T, only Z_0 must be ready at the end of each iteration, while $(Z_{s-1}...Z_1)$ can be generated later. Based on this observation, a new scheduling method is proposed and is shown in Figure 4. In this method,

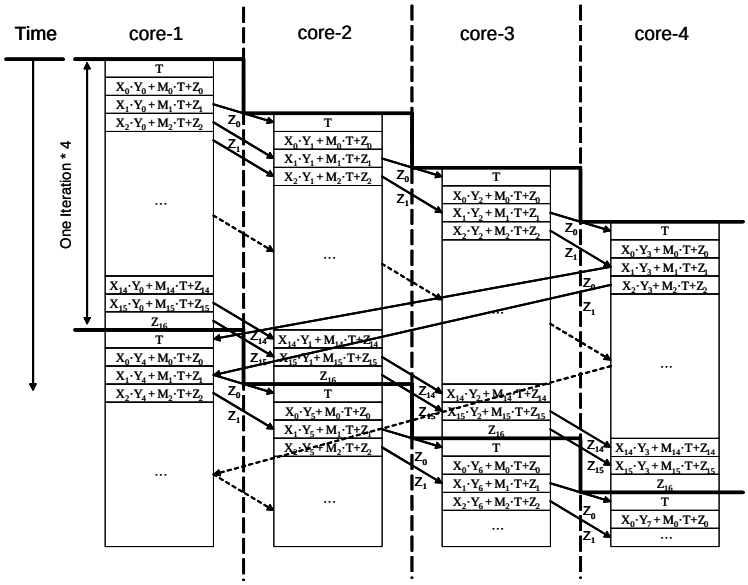

Fig. 3. Instruction scheduling method-I: One iteration is performed with one core. ($n = 256$, $w = 16$, $s = 16$, $N_{arrow} = 240$).

each iteration in Algorithm 1 is performed by multiple cores. Figure 4 shows this scheduling method for a 4-core system. Here we still choose $n = 256$, $w = 16$ and $s = \lceil \frac{n}{w} \rceil = 16$. During the whole loop $(Z_3, .., Z_0)$ is generated and stored in core-1, $(Z_7, .., Z_4)$ in core-2, $(Z_{11}, .., Z_8)$ in core-3 and $(Z_{15}, .., Z_{12})$ in core-4. Carry is only used in the local core. At the end of each iteration, Z_4 is sent to core-1, Z_8 is sent to core-2 and Z_{12} is sent to core-3. After sixteen iterations and a conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is generated and stored separately in four cores. Z can be written to the data memory or can be used by another modular multiplication.

The method-II needs significantly less data transfers between cores than the method-I. T is always generated in core-1, and then distributed to other cores. On a p-core system, $p - 1$ arrows are needed to transfer T in each iteration. To shift Z to the right, $p - 1$ word is transferred, making $p - 1$ arrows in each iteration. As a result, the number of arrows for one modular multiplication is $2(p - 1)s$. When $s = 16$ and $p = 4$, the number of arrows, $N_{arrow} = 96$.

Each arrow in Figure 3 and Figure 4 causes one store and one load operation. The comparison of memory accesses caused by data transfers is presented in Table 2. Here $N_{load-tr}$ and $N_{store-tr}$ are the number of load and store operations caused by data transfers, respectively. $N_{total-tr}$ is the sum of them. Note that in Figure 4 the multiple arrows starting from T cause only one store operation. For the method-II $N_{total-tr}$ is $3ps - 2s$, while $2s^2 - s$ in the method-I. As p is always

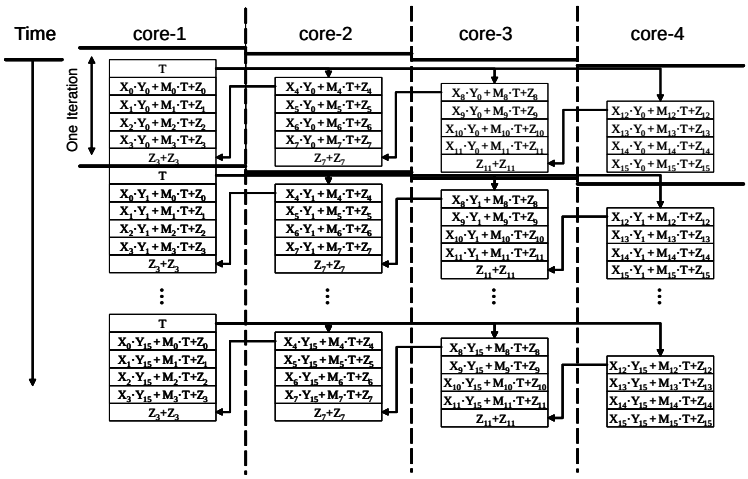

Fig. 4. Instruction scheduling method-II: One iteration can be performed with several cores. ($n = 256$, $w = 16$, $s = 16$, $N_{arrow} = 96$).

much smaller than s, the number of memory accesses caused by data transfers in the method-II is much smaller than that of the method-I.

5 Performance Comparison

Compared to the method-I, the method-II has two major advantages. First, operands and intermediate data are distributed in the register file of each core, thus less registers are required in each core. Second, less data transfers reduce memory accesses, as a result, a single-port data memory can support more cores before becoming the bottleneck. Detailed performance comparison between the method-I and the method-II is given below.

Let N_{mul} be the number of multiplications, N_{add} the number of additions and $N_{load-opr}$ the number of operand load operations. As shown in Table 2, we use $N_{load-tr}$ and $N_{store-tr}$ to denote the number of load and store operations caused by transferring intermediate data, respectively. The total number of memory accesses is denoted as N_{total}. Suppose that Algorithm 1 is implemented, N_{add} is always $4s^2 + s$, and N_{mul} is $2s^2 + s$ regardless the value of p.

Table 2. Number of data memory accesses caused by data transfers.

Scheduling Methods	N_{arrow}	$N_{load-tr}$	$N_{store-tr}$	$N_{total-tr}$
Method-I	$s(s-1)$	$s^2 - s$	s^2	$2s^2 - s$
Method-II	$2(p-1)s$	$2(p-1)s$	ps	$3ps - 2s$

Although N_{mul} and N_{add} are constant, N_{load} and N_{store} vary from different scheduling methods or different number of cores. Since the size of register files has a great influence on the number of memory accesses, it must be taken into account. If the register file is large enough, the operands, X, Y and M can stay in the registers during the whole loop. If not, they may need to be reloaded in each iteration, thus N_{total} becomes larger. Let S_{rf} be the number of entries of w-bit registers in each core's register file. Table 3 shows how the number of load and store operations changes for different size of register files.

Suppose Algorithm 1 is implemented on a w-bit processor. First, if S_{rf} is larger than $3s$, then X, M and Y only need to be loaded to the registers in the beginning of the loop, making $N_{load-opr} = 3s$. Since Z is generated and always stay in the registers in the whole loop, both $N_{load-tr}$ and $N_{store-tr}$ are 0. For $2s < S_{rf} \leq 3s$, only Z and X can be stored in the registers. The $N_{load-opr}$ increases to s^2+2s. For $s < S_{rf} \leq 2s$, only Z can be stored in the registers, thus $N_{load-opr}$ becomes $2s^2+s$. For $S_{rf} \leq s$, X,M, Y_i and Z will be loaded from the data memory in each iteration, making $N_{load-opr} = 2s^2+s$. Z also has to be sent to the data memory in each iteration, which leads $N_{load-tr} = N_{store-tr} = s^2$.

Now suppose that this processor has p general purpose cores. If the method-I is used, each core needs to use X, M and Y_i in each iteration. For $S_{rf} > 2s$, X and M can stay in the registers during the whole loop. In order to load X, M and Y_i it takes $2ps+s$ cycles. In order to transfer Z from one core to another s load operations and s store operations are required. Thus, for s iterations we need $N_{load-tr} = s(s-1)$ and $N_{store-tr} = s^2$ in total. For $s < S_{rf} \leq 2s$, only X can stay in registers during the whole loop, while M and Y_i are reloaded in each iteration. As a result, $N_{load-opr}$ increases to s^2+ps+s. For $S_{rf} \leq s$, X, Y, M and Z need to be reloaded in each iteration, making $N_{load-opr} = 2s^2+s$.

Table 3. The number of cycles required for one Montgomery multiplication for various Register File size (S_{rf}).

Processor type	S_{rf}	$N_{load-opr}$	$N_{load-tr}$	$N_{store-tr}$	N_{total}
Single-core	$S_{rf} > 3s$	$3s$	0	0	$3s$
	$2s < S_{rf} \leq 3s$	s^2+2s	0	0	s^2+2s
	$s < S_{rf} \leq 2s$	$2s^2+s$	0	0	$2s^2+s$
	$S_{rf} \leq s$	$2s^2+s$	$s(s-1)$ *	s^2 *	$4s^2$
Multi-core Method-I	$S_{rf} > 2s$	$2ps+s$	$s(s-1)$	s^2	$2s^2+2ps$
	$s < S_{rf} \leq 2s$	s^2+ps+s	$s(s-1)$	s^2	$3s^2+ps$
	$S_{rf} \leq s$	$2s^2+s$	$s(s-1)$	s^2	$4s^2$
Multi-core Method-II	$S_{rf} > \frac{3s}{p}$	$2s+ps$	$2(p-1)s$	ps	$5ps$
	$\frac{2s}{p} < S_{rf} \leq \frac{3s}{p}$	s^2+ps+s	$2(p-1)s$	ps	$s^2+4ps-s$
	$\frac{s}{p} < S_{rf} \leq \frac{2s}{p}$	$2s^2+ps$	$2(p-1)s$	ps	$2s^2+4ps-2s$
	$S_{rf} \leq \frac{s}{p}$	$2s^2+s$	$s^2+(2p-3)s$ *	s^2+s *	$4s^2+2ps-s$

*Including store and load operations caused by calculating intermediate data.

For the method-II, the memory accesses are less. Since each core keeps a part of X, M and Z, the required register size of each core is less than that of the method-I. To keep X, M and Z in the registers during the whole loop, S_{rf} needs to be larger than $\frac{3s}{p}$. In order to load X,Y and M into registers it takes $2s + ps$ cycles, namely $N_{load-opr} = 2s+ps$. In order to distribute T, one store and $(p-1)$ load operations are needed in each iteration. In order to shift Z one word to the right, $(p-1)$ words of Z must be stored to the data memory and loaded in each iteration. As a result, we need $N_{load-tr} = 2(p-1)s$ and $N_{store-tr} = ps$. For $\frac{2s}{p} < S_{rf} \leq \frac{3s}{p}$, we only keep X and Z in the registers, while load M in each iteration. The $N_{load-opr}$ increases to $s^2 + ps + s$. For $\frac{s}{p} < S_{rf} \leq \frac{2s}{p}$, only Z is stored in registers, thus $N_{load-opr} = 2s^2 + ps$. For the case of $S_{rf} \leq \frac{s}{p}$, all the operands and intermediate data have to be reloaded in each iteration, $N_{load-opr}$ becomes $2s^2 + s$. $N_{load-tr}$ and $N_{store-tr}$ are also increased rapidly. The number of memory accesses in each case is shown in Table 3.

As shown in Table 3, in the case that s and p are fixed, N_{total} changes as various size of register files are used. Note that for the method-II S_{rf} decreases as p increases. To reduce N_{total} for a large s, one can increase S_{rf} and p. However, increasing p doesn't help in the method-I. For example, when $S_{rf} = 16$, $p = 4$ and $s = 64$, N_{total} is 16384 and 16832 for the method-I and method-II, respectively. When $p = 8$, N_{total} is reduced to 10112 for the method-II, while is still 16384 for the method-I. Figure 5 illustrates the comparison.

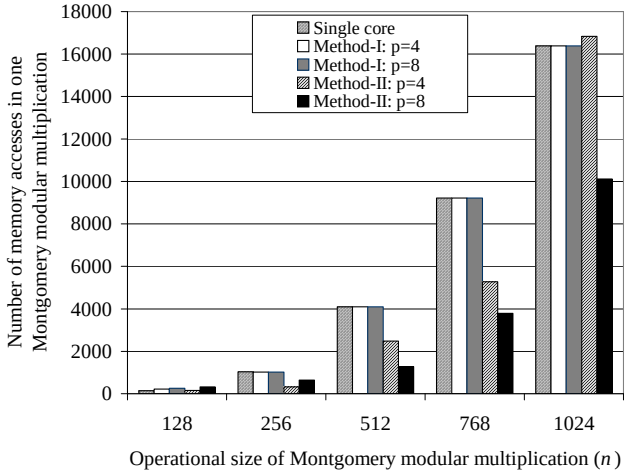

Fig. 5. Number of data memory accesses for various operand bit-length. ($w = 16$, $S_{rf} = 16$).

When the number of cores reaches a specific value, the memory access becomes the bottleneck. Because our proposed architecture uses a single-ported

shared memory, load and store can not be operated in parallel. As a result, the cycles needed by one modular multiplication are no smaller than $N_{load}+N_{store}$. Let $N(s,p)$ be the number of cycles that needed by one n-bit multiplication on a p-core system. Then as p increases, there is a point where $N(s,p) = N_{load}+N_{store}$. After reaching this point, increasing p doesn't improve the performance any more. Because the method-I needs more load and store instructions than the method-II, it reaches this point before the method-II as p increases.

6 Results

The multi-core platform proposed in section 3 is implemented with GEZEL. The GEZEL code is automatically converted to synthesizable VHDL codes. The software program of Montgomery modular multiplication is stored in the instruction memory. The operands, X, Y and M, are stored in the data memory.

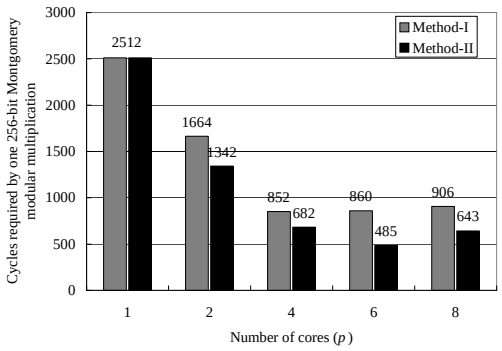

Fig. 6. Performance of 256-bit Montgomery modular multiplication on a multi-core system. ($n = 256$, $w = 16$, $S_{rf} = 16$).

We implemented both method-I and method-II on the platform with various hardware configurations. The results are presented in Figure 6. The results of the method-II show a better performance and a higher scalability in the number of cores compared to that of the method-I. If a single core is used, 2512 cycles are needed to finish one 256-bit Montgomery multiplication. When using 2 cores, 1664 and 1342 cycles are required for the method-I and the method-II, respectively. If 4 cores are used, only 852 cycles are required for the method-I, while 682 cycles are required for the method-II.

On the other hand, when employing more than 4 cores, the performance of the method-I is deteriorated because the number of the memory accesses becomes the bottleneck. For the method-II, the best performance is obtained when $p = 6$ as shown in Figure 6.

Table 4. Performance comparison of modular multiplication.

Reference	**Description**	**Platform**	**Area** (Slices)	**Freq.** (MHz)	**256-bit time**(μs)	**1024-bit time**(μs)
This work (method-I)	2-cores 2 16x16 mults	Xilinx XC2VP30	1102	125	13.3	213.0
	4-cores 4 16x16 mults	Xilinx XC2VP30	2029	125	6.8	131.0
	2-cores 2 32x32 mults	Xilinx XC2VP30	1822	93	4.5	71.6
	4-cores 4 32x32 mults	Xilinx XC2VP30	3173	93	2.6	44.0
This work (method-II)	2-cores 2 16x16 mults	Xilinx XC2VP30	1102	125	10.7	189
	4-cores 4 16x16 mults	Xilinx XC2VP30	2029	125	5.5	134.7
	2-cores 2 32x32 mults	Xilinx XC2VP30	1822	93	3.7	64.0
	4-cores 4 32x32 mults	Xilinx XC2VP30	3173	93	2.2	33.0
Tenca & Koç [6]	Software implementation	ARM processor	-	80	43	570
Cohen *et al.* [20]	Software implementation	UltraSPARC GMP library	-	143	14.6†	–
Itoh *et al.* [21]	Software implementation	DSP TMS320C6201	-	200	2.68‡	–
Brown *et al.* [22]	Software implementation	Pentium II	-	400	1.57§	–
Sakiyama *et al.* [23]	CSAs based Dual-Field	Xilinx XC2VP30	4836	110.4	0.80	–
Kelley *et al.* [24]	4-PEs 8 16x16 mults	Xilinx XC2V2000-6	360*	135	0.68	8.3
Mentens [7]	34 16x16 mults	Xilinx XC2VP30	1927	73	0.27	–
Mentens [7]	130 16x16 mults	Xilinx XC2VP30	7244	64	0.31	1.07

* Author's estimation from the original paper.
† 224-bit normal modular multiplication.
‡ 239-bit Montgomery modular multiplication.
§ Using fixed modulo for fast reduction.

For the purpose of checking the maximum frequency, the platform is implemented on Xilinx Virtex-II PRO (XC2VP30) FPGA. A maximum frequency of 125 MHz could be achieved if 16-bit cores ($w = 16$) are used. For 32-bit cores ($w = 32$), a maximum frequency of 93 MHz can be obtained. The instruction

memory and the data memory are implemented in the block RAM on the FPGA board. The number of slices here only includes main controller and cores. The performance comparison between our software implementations and the state-of-the-art implementations is summarized in Table 4.

As shown in Table 4, our software implementation can achieve high speed and good scalability. Taking the method-II as an example, when using two 32-bit cores, the 256-bit modular multiplication is almost 10 times faster than the implementation on the ARM processor [6] and almost 4 times faster than the implementation on the UltraSPARC processor [20]. When using four 32-bit cores, the implementation of 256-bit modular multiplication is as fast as the implementation on TI's DSP (TMS320C6201) [21], which can issue eight 32-bit instructions in parallel. The implementation in [22] is fast, however it only supports fixed modulus. Compared to the state-of-the-art hardware implementations [23, 24, 7], software implementations are still much slower. However, hardware implementations add area and complexity to the whole system, and have far less flexibility than software implementations.

7 Conclusions

In this paper, we introduced an efficient software implementation of the Montgomery multiplication algorithm on a multi-core system. A prototype of general multi-core systems is implemented. Our newly proposed scheduling method could reduce the number of data transfers between different cores. As a result, the performance of 256-bit Montgomery multiplication was improved by a factor of 1.87 and 3.68 when using 2-core and 4-core systems, respectively.

Our future work includes a hardware implementation based on our proposed parallel-processing algorithm with a special data-path that can perform multiple arithmetic operations. A software implementation of this algorithm on commercial multi-core processors is also in progress. We believe that the scheduling method proposed in this paper can achieve high flexibility and high-performance in both software and hardware implementations.

Acknowledgments

Junfeng Fan and Kazuo Sakiyama are funded by a research grant of the Katholieke Universiteit Leuven and FWO projects (G.0450.04, G.0475.05). This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by the EU IST FP6 projects (SESOC and ECRYPT), by the K. U. Leuven, and by the IBBT-QoE project of the IBBT.

References

1. R. L. Rivest, A. Shamir and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978.

2. N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203-209, 1987.
3. V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in LNCS, pages 417-426. Springer-Verlag, 1985.
4. P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*,44:519-521,1985.
5. S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*,42(6):693-699,June 1993.
6. A. Tenca and Ç. K. Koç. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Transactions on Computers*, 52(9):1215-1221, September 2003.
7. N. Mentens, Secure and efficient coprocessor design for cryptographic applications on FPGAs. PhD Thesis, June, 2007.
8. http://download.intel.com/products/processor/xeon/dcprodbrief.pdf
9. http://download.intel.com/products/processor/xeon/dc53kprodbrief.pdf
10. Y. Kanno, H. Mizuno, Y. Yasu, K. Hirose, Y. Shimazaki, T. Hoshi, Y. Miyairi, T. Ishii, T. Yamada, T. Irita, T. Hattori, K. Yanagisawa, and N. Irie. Hierarchical Power Distribution with 20 Power Domains in 90-nm Low-Power Multi-CPU Processor. *ISSCC Dig. Tech. Papers*, pages 540-541, Feburary 2006.
11. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html
12. C. D. Walter. Montgomery's exponentiation needs no final subtraction. *Electronic letters*, 35(21):1831-1832, October 1999.
13. Ç. K. Koç, T. Acar and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*,16:26-33,1996.
14. M. E. Kaihara and N. Takagi. Bipartite modular multiplication. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2005*, number 3659 in Lecture notes in Computer Science, pages 201-210, September 2005. Springer-Verlag.
15. K. Iwamura, T. Matsumoto, and H. Imai. High-speed implementation methods for RSA scheme. In R. A. Rueppel, editor, *Advances in Cryptology: Proceedings of EUROCRYPT 92*, number 658 in Lecture Notes in Computer Science, pages 221-238. Springer-Verlag, 1992.
16. T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. *In Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 70C77, Adelaide, Australia, April 14-16 1999.
17. L. Batina and G. Muurling. Montgomery in practice: How to do it more efficiently in hardware. In B. Preneel, editor, *Proceedings of RSA 2002 Cryptographers Track*, number 2271 in Lecture Notes in Computer Science, pages 40-52, San Jose, USA, February 18-22 2002. Springer-Verlag.
18. S. H. Tang, K. S. Tsui and P. H. W. Leong. Modular exponentiation using parallel multipliers. *Proceedings of the 2003 IEEE International Conference on Field Programmable Technology (FPT)*, Tokyo, 52-59. 2003
19. P. Schaumont and I. Verbauwhede, Interactive cosimulation with partial evaluation. *Proceedings of Design Automation and Test in Europe (DATE 2004)* pp. 642-647, 2004.
20. H. Cohen, A. Miyaji and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. *Asiacrypt'98*, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.
21. K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara: Fast implementation of public-key cryptography on a DSP TMS320C6201. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES'99*, LNCS 1717, pp. 61-72, Springer-Verlag, 1999.

22. M. Brown, D. Hankerson, J. López and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. *Topics in Cryptology, CT-RSA 2001*, LNCS 2020, pp. 250-265, Springer-Verlag, 2001.
23. K. Sakiyama, B. Preneel and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware implementation. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, pages 787-790, 2006.
24. K. Kelley and D. Harris. Parallelized very high radix scalable Montgomery multipliers. *Conference on Signals, Systems and Computers*, pages 1196-1200, 2005.