Synthesis of Parallel Synchronous Software

Pantea Kiaei, *Student Member, IEEE,* Patrick Schaumont, *Senior Member, IEEE*

***Abstract*—In typical embedded applications, the precise execution time of the program does not matter, and it is sufficient to meet a real-time deadline. However, modern applications in information security have become much more time-sensitive, due to the risk of timing side-channel leakage. The timing of such programs needs to be data-independent and precise. We describe a parallel synchronous software model, which executes as N parallel threads on a processor with word-length N. Each thread is a single-bit synchronous machine with precise, contention-free timing, while each of the N threads still executes as an independent machine. The resulting software supports fine-grained parallel execution. In contrast to earlier work to obtain precise and repeatable timing in software, our solution does not require modifications to the processor architecture nor specialized instruction scheduling techniques. In addition, all threads run in parallel and without contention, which eliminates the problem of thread scheduling. We use hardware (HDL) semantics to describe a thread as a single-bit synchronous machine. Using logic synthesis and code generation, we derive a parallel synchronous implementation of this design. We illustrate the synchronous parallel programming model with practical examples from cryptography and other applications with precise timing requirements.**

***Index Terms*—repeatable-time programming, data-independent timing, bitslicing, automatic code generation**

I. Introduction

Producing software with precise, repeatable timing is a challenging task. First, the software application itself may have data-dependent processing complexity, such as with data-dependent loops. Second, the execution time of the application on the processor may be affected by the memory hierarchy and the run-time state of the processor. Third, the timing of the execution may be affected by resource contention when several parallel threads share the same processor resource. Among these three problems, the second and the third are most difficult because they are outside of the control of the programmer. In cryptographic applications, data-dependent timing variations may be exploited as timing side-channel leakage, either directly as an effect of data-dependent control flow, or indirectly as an effect of contention on shared processor resources. To avoid timing side-channels, we need data-independent timing.

In this contribution, we propose a programming model that yields these timing characteristics. We contrast our proposal with earlier work towards precise software timing for embedded applications, PRET [1], [2]. A fundamental idea of PRET is to use instruction scheduling to avoid resource contention in the processor in the pipeline. By spacing the instructions of timing-critical threads several cycles apart, stall-free execution is achieved in the pipeline. As a consequence, the timing of individual threads is repeatable regardless of the processor state. To ensure overall processor utilization, PRET combines multiple timing-critical threads with time-interleaving and a customized instruction scheduling technique [2].

This work was supported by National Institute of Standards and Technology (NIST) under grant 70NANB17H280.

The authors were with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061 USA. (E-mail: pantea95@vt.edu; schaum@vt.edu)

Our insight in this paper is that such time-sensitive threads can also be combined *spatially* within a processor word, instead of *temporally* using interleaved instruction streams. The advantage of spatially combining the threads (instead of using time-based interleaving) is that we don't need to adapt the processor for interleaved instruction execution. To implement a spatial arrangement of threads, we organize each thread as a single-bit program, and execute the overall application as a vectorized version of the single-bit program. We emphasize that the proposed model goes beyond software bit-slicing [3], which is strictly functional and ignores the control flow and the state within each slice.

To simplify the development of single-bit programs, we adopt a synchronous execution model. A single-bit program is captured as a synchronous Finite State Machine with Datapath (FSMD), and the execution of this program follows a sequential schedule of the bit-operations that define the FSMD. The vectorized form of the single-bit program is then achieved with bitwise instructions over the processor word. The vectorized form is a *parallel* synchronous program. Since each thread has its own state, each thread executes as an independent FSMD. However, the instruction count for one iteration of the overall program is constant and repeatable, and therefore the execution time of these FSMD threads becomes repeatable too. A prototype implementation of a synthesis tool starts from a Verilog input specification and generates C code with inline assembly optimized for an embedded target. We demonstrate several useful examples of parallel synchronous programming (PSP).

The outline of this contribution is as follows. Section II develops the cardinal components of parallel synchronous programming. Section III discusses an example and proposes a code generation methodology. Section IV describes experimental results. Section V concludes the paper.

II. Preliminaries

We develop a software execution model that leads to repeatable, data-independent timing. We first define what is meant by repeatable and data-independent timing in software. We then describe software bitslicing, which can offer such timing characteristics for functions (i.e. straight-line stateless programs). Next, we explain how to extend the semantics of software bitslicing from straight-line programs to synchronous FSMD. The result is a Parallel Synchronous Program (PSP).

A. Desired Timing Properties

Programs written as PSP aim for repeatable timing as well as data-independent timing. The former is useful in real-time embedded software design, while the latter is useful for secure systems design. We motivate and differentiate each property.

Edwards *et al.* make a distinction between repeatable timing and predictable timing [4]. Repeatable timing means that every correct execution of a program uses the same timing. Repeatable timing is desired as a property of the program, not of the program running on a specific processor. Repeatable timing is needed in the context of real-time applications when timing jitter is a concern. For example, when a physical sensor must be read from software at a specific sample rate, then the software needs to have repeatable timing. Jitter is typically caused by resource contention and interrupts.

A second relevant domain for PSP is that of secure software. In recent years, a rich collection of attacks have been found to exploit the *implementation* characteristics of secure software rather than the program logic itself. The best known of these are side-channel attacks and micro-architectural attacks, which rely on precise execution time measurement [5]. To thwart these attacks, software with (secret-)data-independent timing is needed. This is hard because modern micro-architectures are rife with architectural contention and context-dependent timing. Even if there are no obvious dependencies in the program logic, there may still be hidden dependencies in the micro-architecture. The cryptographic community is well aware of the risk of timing-based side-channel leakage, leading to the design of so-called *constant-time* software that avoids data-dependencies in the program execution time [6]. The resulting programs are not literally constant-time, but rather they adopt data-independent control flow and memory access patterns.

We argue that software written as a Parallel Synchronous Program (PSP) provides repeatable timing as well as data-independent timing. PSP achieves these properties by combining two concepts: software bitslicing and synchronous FSMD. The following subsections introduce both.

B. Bitslicing

Software bitslicing was originally proposed for high throughput software implementations [3]. In this model of programming, a program is expanded into 1-bit (Boolean) operations as follows. A $k-$bit variable with bits $b_{k-1}...b_1b_0$ is distributed over k registers $R_{k-1}...R_1R_0$, such that register R_i holds bit b_i. An $N-$bit processor operates as an $N-$way SIMD processor, processing N instances of the $k-$bit variable in parallel, and storing these instances in k registers. Bitsliced programs are Boolean programs written with bit-wise logic operations. The rationale of bitslicing is that it guarantees full utilization of the processor word-length. The absence of control flow ensures that each iteration through a bitslice function uses the same amount of instructions. In addition, the absence of state (memory) in a bitslice function eliminates cache timing effects. For this reason, bitslicing is often applied in the context of developing programs that are *constant-time* (in the cryptographic sense). However, software bitslicing is insufficient as a general-purpose methodology for software. Because bitslice functions do not have control flow, control operations are typically emulated using non-bitsliced logic surrounding bitsliced expressions. This prevents individual slices from operating as independent threads of control. Bitslice programming essentially applies only to functions. The management of the program state resides outside of the bitslice logic.

C. Synchronous FSMD

We next describe how to introduce control flow and state into software bitslicing. A Boolean (1-bit) program does not

```
while true do
    wait for clock_tick;
    outputs = eval(inputs, current_state);
    next_state = update(inputs, current_state);
    current_state = next_state;
end
```

Fig. 1. Basic structure of a synchronous program

offer the concept of an address space or control flow instructions. Therefore, we propose introducing the control flow through the intermediary of a synchronous FSMD. FSMDs are common in digital hardware design, and they are routinely applied in register-transfer-level designs. An FSMD is a synchronous model of computation combining a datapath and a finite-state controller. Computations are done on the datapath under control of the FSM [7]. Each synchronous clock cycle, the FSM computes a single state transition and selects one or more operations in the datapath. The execution of datapath operations depends on the current state of the FSM, and the state transition conditions in the FSM depend on the current state of the datapath. Conditional control flow is expressed using dataflow-like semantics: the datapath will compute both the true and false case of the control condition, and the correct result will be selected using multiplexing.

To map a synchronous FSMD into software, we adopt a synchronous execution model as shown in Figure 1. Every loop in this program corresponds to a single clock cycle of the synchronous FSMD model. The software awaits the occurrence of a clock tick to read all inputs and evaluate all outputs concurrently [8]. The `eval()` function in Figure 1 computes the FSM next-state as well as the datapath next-state. The `update()` function adjusts the current state of the FSM and the datapath to the next state. State update is handled synchronously: each state variable in the synchronous FSMD is split into two copies, the *current state* and the *next state*. This avoids race conditions, and ensures that the program will always compute the same result regardless of the scheduling of `eval()` and `update()`. The PSP is implemented as N parallel copies of the program of Figure 1, where every thread is tied to the same global clock tick, and each thread is a 1-bit program expressed as a synchronous FSMD.

III. Synthesis of Parallel Synchronous Software

We next demonstrate how to create PSP. We first describe the example PSP design of a parallel greatest-common-divisor program, and next discuss a design flow that synthesizes PSP software from a synchronous FSMD description.

a) Example: Figure 2a shows the outline of a 4-bit GCD module. We express the functionality of the GCD algorithm as an FSMD model. After a `start` control pulse, the module reads two 4-bit inputs `a` and `b`, and repeatedly subtracts the smaller value from the larger value until they are equal. A `done` pulse is generated to indicate completion of the algorithm. A two-state control FSM drives the loading of two 4-bit registers `a` and `b` and their iterative computation.

A PSP version of the GCD algorithm for a 32-bit processor executes 32 parallel copies of the GCD. We create this software by converting the FSMD to a gate-level netlist using logic synthesis. We target a generic technology with a logically complete set of primitive functions (such as `AND`, `OR` and

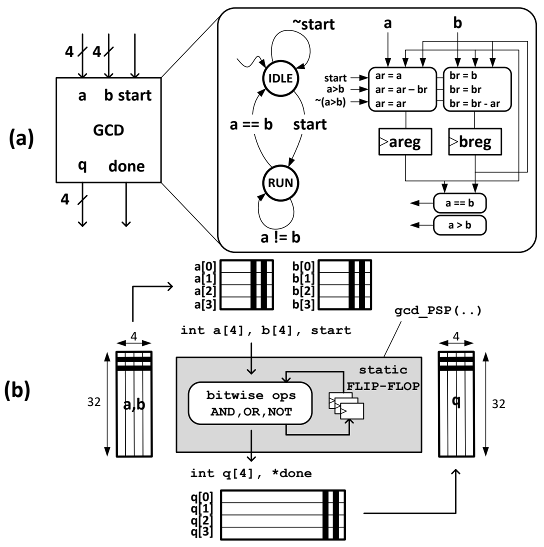

Fig. 2: GCD (a) Interface and (b) FSMD model

NOT) as well as a storage element such as a flip-flop (Figure 2b). The outcome of the logic synthesis is a netlist in terms of logic elements. We then rewrite the netlist as a sequential function by leveling the netlist according to data dependencies from input to output. The logic cells are replaced by bit-wise operations, and the flip-flops are replaced by `static` (or global) variables. The resulting function declaration is as follows.

```
gcd_PSP(int a[4], int b[4], // data input
        int q[4],           // data output
        int start,          // control in
        int* done);         // status out
```

Each invocation of this function corresponds to a single synchronous iteration (one clock cycle of the synchronous FSMD). An important difference between the circuit of Figure 2a and the PSP function in Figure 2b is the degree of parallelism; The circuit in Figure 2 computes a single GCD whereas the `gcd_PSP` function is a software design that computes 32 *concurrent* GCD algorithms independently, each with their own `start` and `done` bits. The inputs and outputs of `gdc_PSP` are in bitsliced form. For example, `a[2]` contains the second bit of 32 different inputs. Hence, a call to `gcd_PSP` needs to transpose the input and output arguments.

b) An Automated Flow: We implemented a software synthesis flow for PSP that starts from an FSMD description in a Verilog program. An open-source Verilog synthesis tool [9] converts the FSMD into a netlist in terms of generic target technology for Boolean logic and a state element. The target library for logic synthesis is adjusted in function of the targeted processor. Table I demonstrates a sample mapping for several embedded processors. The state elements (flip-flop) are mapped to `static` variables.

The netlist is then converted to software as follows. The netlist is topologically sorted, from the primary inputs and flip-flop outputs to the primary outputs and the flip-flop inputs. Next, each primitive gate is converted to a bitwise operation which is either emulated in C or else added through inline assembly. We rely on the C compiler to create a sequential schedule for the gate netlist that will minimize the register pressure on the processor. The following section applies the automated flow on several examples.

TABLE I: Instructions targeted by PSP synthesis

processor	**suitable instructions for PSP**
ARM Cortex-M4	AND, BIC, EOR, MOV, MVN, ORN, ORR
RISC-V	AND, OR, XOR
MSP430	AND, BIC, BIS, XOR
AVR	AND, COM, EOR, OR

TABLE II: Evaluated encryption ciphers and comparison of performance of the PSP and normal implementations of them

	cipher properties				**speed (cycles/byte)**		
cipher	**block size**	**key size**	**rounds**	**type**	**PSP**	**normal**	**speedup**
SIMON	128	128	68	Feistel	744.48	1315.63	1.7×
PRESENT	64	80	31	SPN	399.61	1069.06	2.6×
LED	64	64	32	SPN	-	-	-
Midori	64	128	16	SPN	236.90	2233.38	9.4×

IV. Experimental Results

We analyze our flow and the resulting performance using several examples. We target the 48 MHz ARM Cortex-M4F processor, which comes with the Texas Instruments MSP432P401R Launchpad and implements the ARMv7E-M architecture. Table III summarizes our results. The numbers reported on this table are compiled with size optimization (`-Os`).

The first two examples, GCD and PWM, illustrate the general-purpose nature of PSP as well as its real-time characteristics. For these examples, Table III lists the number of processor clock cycles per synchronous cycle. Computing 32 parallel GCD's thus takes 382 clock cycles per synchronous cycle, *i.e.*, per iteration of the GCD while-loop.

The Pulse Width Modulator (PWM) generates pulses with a fixed period while having different duty cycles. The PSP version of this function in a 32-bit architecture can generate 32 pulses with varying cycles of duty at the same time. Our implementation demonstrates a PWM with 8-bit resolution. The synchronous cycle of our PWM uses 239 ARM cycles, which provides a minimum pulse width of $\frac{239}{48\text{MHz}} = 4.98\mu s$ and a period of $2^8 \times \frac{239}{48\text{MHz}} = 1,275\mu s$ or 784Hz.

The second group of examples are taken from cryptography [10]–[13]. Their characteristics are summarized in Table II. SIMON 128/128 is a block cipher with the Feistel structure and consists of 68 calls to the same round encryption routine. We used two different realizations of SIMON, the first one with a bit-parallel data-path and the second one with a bit-serial data-path [14]. In traditional hardware design, bit-serial methodologies are used to minimize area footprint at the expense of throughput. In the PSP execution model of software, we expect the lower gate-count of a bit-serial input specification to translate to fewer bit-wise operations in the program, and hence to a smaller code footprint. Further, we expect the bit-serial PSP design to have a lower throughput due to the lower computational effort done per synchronous clock cycle.

The first part of Table III shows that the models are small enough to fit on a simple embedded architecture. Furthermore, we observe, similar to their hardware designs, the bit-serial implementation of SIMON is 20% smaller than its bit-parallel counterpart in code size, whereas the bit-parallel version is $40\times$ faster and has a higher throughput than the bit-serial version. The second part of Table III shows the overhead of data movements. The overhead values reported are calculated as *the*

TABLE III: Evaluation of parallel synchronous examples on 48MHz Cortex-M4F processor

	performance and cost			**instructions breakdown**									
example	**number of cycles (32 parallel runs)**	**throughput (Kbps)**	**code size (Kb)**	**AND**	**ORR**	**BIC**	**EOR**	**ORN**	**MVN**	**MOV**	**STR**	**LDR**	**overhead**
GCD	382	-	11.88	28	34	6	9	7	8	21	28	73	54.46%
PWM	239	-	11.82	29	20	11	6	2	8	0	23	39	44.93%
SIMON bit-parallel	381,175	515.79	23.40	907	470	180	367	27	4	18	1033	2002	60.96%
SIMON bit-serial	15,370,190	12.79	18.81	854	313	23	16	19	13	7	593	686	50.95%
PRESENT	102,301	960.93	17.79	226	282	60	119	70	32	24	454	861	62.92%
LED	139,949	702.43	20.29	379	301	80	395	60	60	138	556	1258	60.49%
Midori	60,646	1620.95	18.28	336	265	60	242	124	78	91	438	930	56.90%

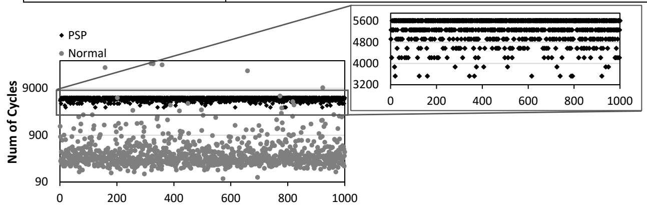

Fig. 3: Runtime of normal and PSP implementations of the GCD algorithm on 1000 random inputs.

number of move instructions (MOV, STR, LDR) divided by the total number of instructions. Moving the data takes about 45-60% of the entire instructions, which is expected for a straight-line program. For comparison, the data-moving overhead for a regular (non-bitsliced) implementation of SIMON on NEON in the SUPERCOP benchmark [15] is 34%.

We compare our PSP designs of cryptographic ciphers with their available normal implementations in Table II. In the CRYPTREC lightweight project [16], SIMON-128/128 and Midori-64 ciphers are implemented in software for the RL78 16-bit microcontroller. The throughputs of the PSP implementation of these ciphers in this work are respectively almost 1.7× and 9.4× higher. PRESENT-80 is evaluated in the FELICS [17] project on ARM Cortex-M3. Even though the implementation of PRESENT-80 in FELICS uses pre-computed keys, still the runtime of our PSP implementation of this cipher plus its key generation is approximately 2.6× smaller. Furthermore, to show the repeatable-timing property of PSP, we compare the runtime of the PSP and non-PSP implementations of GCD calculator for 1000 random inputs. As shown in Figure 3, the PSP implementation has a quantized runtime (with steps of length the runtime of one PSP function) whereas the runtime of the normal GCD function varies with an average of 580.475 and a standard deviation of 1969.29 clock cycles.

V. Conclusion

We presented parallel synchronous programming as a high-throughput, fixed-time model of programming, which is beneficial in safety-critical applications. We introduced an automated method for PSP code generation that can be implemented without any dependency on commercial tools. The PSP generation can be customized for the target processor to have a better performance by defining custom libraries. Finally, through examples and discussions, we demonstrated the potential of parallel synchronous software.

References

[1] E. A. Lee, J. Reineke, and M. Zimmer, "Abstract PRET machines," in *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*. IEEE Computer Society, 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1109/RTSS.2017.00041

[2] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*. IEEE Computer Society, 2014, pp. 101–110. [Online]. Available: https://doi.org/10.1109/RTAS.2014.6925994

[3] E. Biham, "A fast new DES implementation in software," in *International Workshop on Fast Software Encryption*. Springer, 1997, pp. 260–272.

[4] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *27th International Conference on Computer Design, ICCD 2009, Lake Tahoe, CA, USA, October 4-7, 2009*. IEEE Computer Society, 2009, pp. 54–59. [Online]. Available: https://doi.org/10.1109/ICCD.2009.5413177

[5] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6

[6] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[7] P. R. Schaumont, *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.

[8] P. Caspi, S. Tripakis, and P. Raymond, "Synchronous programming." 2007.

[9] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[10] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[11] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[12] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The led block cipher," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2011, pp. 326–341.

[13] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: A block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2015, pp. 411–436.

[14] A. Aysu, E. Gulcan, and P. Schaumont, "SIMON says: Break area records of block ciphers on FPGAs," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 37–40, 2014.

[15] "Implementation of SIMON and SPECK lightweight block ciphers for the SUPERCOP benchmark toolkit," https://github.com/nsacyber/simon-speck-supercop/tree/master/crypto_stream/simon128128ctr/neon, accessed: 10-12-2019.

[16] CRYPTREC Lightweight Cryptography Working Group, *CRYPTREC Cryptographic Technology Guideline (Lightweight Cryptography)*, CRYPTREC Report March 2017.

[17] D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov, "Triathlon of lightweight block ciphers for the internet of things," *Journal of Cryptographic Engineering*, vol. 9, no. 3, pp. 283–302, 2019.