Architectural Time-predictability Factor (ATF): A Metric to Evaluate Time Predictability of Processors

Yiqiang Ding, Wei Zhang
Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA 23284
wzhang4@vcu.edu

***Abstract*—Due to the prohibitive cost of worst-case timing analysis for modern processors, the design of time-predictable processors has become increasingly important for hard real-time and safety-critical systems. However, to the best of our knowledge currently there is no effective and widely accepted metric to quantitatively evaluate time predictability of processors, which greatly impedes the advancement of time-predictable processor design.**

This paper first introduces the concept of architectural time predictability (ATP), which separates timing uncertainty concerns caused by hardware from software. We then propose a metric called Architectural Time-predictability Factor (ATF) to measure architectural time predictability. Our evaluation on a Very Long Instruction Word (VLIW) processor indicates that ATF is an effective metric to quantitatively evaluate architectural time predictability of a whole processor as well as its architectural and microarchitectural components such as caches, branch prediction, speculative execution, parallel pipelines, and Scratch-Pad Memory (SPM). Thus ATF can be used to quantitatively guide architectural design for enhancing time predictability or making better trade-offs between performance and time predictability.

I. Introduction

As well known, accurately estimating the worst-case execution time (WCET) is crucial for hard real-time and safety-critical systems. However many traditional microprocessor architectural designs such as caches and branch prediction are aimed at improving the average-case performance, which unfortunately are harmful to time predictability [1], [2]. As a result, WCET analysis for modern processors has become very complex, if not impossible. The recent development of multithreaded and multicore architectures aggravates this problem. The resource contention in those architectures can adversely affect the execution time and further complicate WCET analysis. On the other hand, designing a microprocessor with high time predictability but low performance is likely to be useless. Therefore researchers have been studying time-predictable microprocessor design to reconcile time predictability and performance [1], with the goal to achieve better time predictability (or WCET analyzeability) while minimizing the impact on average-case performance.

Some designs of time-predictable processors have been proposed. Delvai et al. designed SPEAR (Scalable Processor for Embedded Applications in Real-Time Environments), which employed a simple 3-stage pipeline and no cache memories [3]. Paolieri et al. [4] examined a time-predictable multicore architecture to support WCET analyzeability. Colnaric et al. [1] proposed a simple asymmetrical multiprocessor architecture for hard real-time applications, in which no dynamic architectural feature such as pipelines and caches was used. Yamasaki et al. [5] studied prioritized multithreaded processor through IPC control and prioritization. Edwards and Lee [6] proposed the precision timed (PRET) machine. Schoeberl [7] proposed a time-predictable Java processor. However, in all these studies, time predictability was not quantitatively evaluated, probably due to the lack of an effective and widely accepted metric when these studies were conducted.

Compared to the quantitative study of microprocessor design for improving the average-case performance, the time-predictable processor design so far has been a qualitative study and ad-hoc somehow. Because there is no well-defined metric to evaluate time predictability of processors, most prior work on time-predictable processor design either simply reported the worst-case performance through measurement or analysis, or qualitatively explained their designs were time-predictable by removing undesirable architectural features. The lack of a metric of time predictability thus not only prevents designers from understanding and comparing different time-predictable designs quantitatively, but also makes it difficult to make intelligent trade-offs between time predictability and average-case performance, which often conflict with each other. To make an analogy, it would be hard to imagine how much progress the computer architecture community would have made without having a metric to quantitatively evaluate average-case performance!

Lately, defining a metric of time predictability has received considerable attention by the real-time and embedded computing community. To the best of our knowledge, Thiele et al. [1] defined time-predictability as the pessimism of WCET analysis and BCET analysis. Grund [8] defined time-predictability as the relation between BCET and WCET and argued that time predictability should be an inherent system property. Grund et al. [9] then proposed a template for predictability definitions and refined the definition into state-induced time predictability and input-induced time predictability. Kirner and Puschner [10] formalized a universal definition of time predictability by combining WCET analyzeability and the stability of the

system. However, in all the above work except Grund et al. [8], [9], the calculation of time predictability is still dependent on the computation of WCET. Since the WCET estimation is usually pessimistic and there is no standard way to compute WCET (though different methods to derive WCET such as abstract interpretation and static cache simulation etc. do exist), any time predictability metric relying on WCET analysis is likely to be inaccurate and hard to be standardized in practice.

Moreover, in all the above works except Grund et al. [8], [9], the definition of time predictability does not separate the time variation caused by software and hardware, making it overly complicated to derive a time predictability metric that can effectively guide the architectural design for time predictability. While Grund et al. [8], [9] proposed state-induced timing predictability (SIP) to separate timing uncertainty between hardware and software, the metric they proposed to evaluate SIP needs to exhaustively find out the maximum and minimum execution time of all different states, which may not be computationally feasible. In contrast, this paper proposes a metric to efficiently assess architectural time predictability, and its effectiveness has been validated on a Very Long Instruction Word (VLIW) processor.

In this paper, we make the following contributions to the time-predictable design of processors:

1) We introduce the concept of timing contract and architectural time predictability (ATP) to separate the timing unpredictability concern caused by hardware design from software, thus making it feasible to quantitatively assess and guide time-predictable architectural design.
2) We propose to use Architectural Time-predictability Factor (ATF) as a metric to quantitatively evaluate architectural time predictability of a processor, as well as architectural time predictability of various architectural and microarchitectural components of the processor.
3) We have evaluated the ATF of a VLIW processor as well as its microarchitectural components, including caches, parallel pipelines, branch predictor, speculative execution and the use of SPM. To the best of our knowledge, this is the first paper to use a quantitative metric to systematically evaluate the time predictability of a high-performance processor.

The remaining of the paper is organized as follows. Section II introduces the concept of architectural time predictability. Section III defines the metric of architectural time-predictability factor. Section IV qualitatively analyzes architectural time predictability of a VLIW processor. The evaluation methodology and the experimental results are presented in Section V and Section VI respectively. Finally, the conclusions are made in Section VII.

II. Architectural Time Predictability

While static timing analysis aims at estimating the WCET safely and as close as possible to the actual WCET of a given processor, whether it is time-predictable or not; the goal of time-predictable architectural design is to design processor architectures so that their timing behavior can be precisely and efficiently predicted. To predict the timing behavior of a processor, we must have a desirable baseline timing behavior to compare with. This baseline time behavior is called the **timing contract** in this paper, as it functions like a contract to guide the timing behavior of the actual execution. For example, the timing contract may specify how many cycles each instruction takes, in which order instructions can overlap their execution in the pipelines etc. If a processor is designed and implemented to fully enforce the timing contract, then it will be fully architecturally time-predictable. Therefore, we can then define architectural time predictability as the following.

Definition 1: **Architectural Time Predictability**: Given an architectural design of a processor, architectural time predictability indicates how close the actual timing behavior is to the baseline timing behavior specified in the timing contract of the processor.

Since not all the architectural designs are fully time-predictable, how do we specify the timing contract for an architectural component that is inherently not time-predictable? In that case, *the timing contract should specify the desired timing behavior while also ensuring high performance*. In other words, optimistic, not pessimistic assumption is preferred to establish an "ideal" baseline processor. For example, if a processor employs a cache memory, the desired timing behavior should be all cache hits, i.e. a perfect cache. While assuming all cache misses is still time-predictable, the performance will be too bad and hence is not desirable. On the other hand, when the timing behavior of an architectural component is totally time-predictable, no assumption, whether optimistic or not, should be made to objectively model the actual timing behavior. For example, if a processor employs a scratch-pad memory, then the latency of every load instruction is fixed and known (i.e. the data are either from the SPM or from the memory). Therefore, the timing contract of this processor should specify the latencies of all the loads without making further assumption.

It should be noted that ATP is independent of the timing uncertainty caused by software. If the input changes, a different path is exercised and the execution time can vary, but this processor can be still fully time-predictable if the execution time exactly follows the timing contract (i.e. the timing variation caused by different inputs is the same for both the "ideal" processor specified in timing contract and the real processor). In other words, the goal of time-predictable processor design should not be to ensure the execution time is not varied or can be bounded with different inputs. Bounding the worst-case execution time with various inputs should be the business of WCET analysis, not the hardware design. However, a time-predictable processor can make WCET analysis in general and the low-level analysis in particular significantly easier as the impact of microarchitectural components (e.g. caching, branch prediction) on the execution time can be predicted or controlled.

III. Architectural Time-predictability Factor

Built upon the definition of ATP, we propose to use **Architectural Time-predictability Factor** to quantitatively evaluate *architectural time predictability*. Given a processor P, an arbitrary real-time trace T that is a stream of instructions with a fixed input, the actual dynamic execution time $D(P,T)$, and the statically predicted execution time based on the timing contract $S(P,T)$, ATF is defined as the following.

$$ATF(P,T) = \frac{D(P,T)}{S(P,T)} \tag{1}$$

It should be noted that here we evaluate ATP based on an arbitrary trace. Given different inputs, a real-time program may generate different traces, thus ATF for this program can be computed based on the ATFs of different traces, for example as an average or standard deviation of the ATFs for all the traces evaluated. This is very similar to performance evaluation, in which we can get the execution time of each trace, and derive an average performance result across different traces to indicate the overall performance. Note that the execution time variation due to different inputs or traces are caused by software unpredictability. Techniques to analyze the worst-case program paths have been extensively studied in the literature of WCET analysis [2], which is complementary to the architectural time predictability studied in this paper. To simplify discussion, we focus on studying ATF for an arbitrary trace in the rest of the paper.

Given a trace T, $D(P,T)$ can be measured at runtime. Thus the remaining question is how we calculate $S(P,T)$. While $S(P,T)$ can be computed statically, it is quite different from static timing analysis, as we cannot require the processor to always produce the worst-case performance to make itself time-predictable. The $S(P,T)$ is statically computed according to the timing specification defined in the timing contract. Since the timing contract specifies the timing behavior of an architecture that is fully time-predictable, $S(P,T)$ should be independent of the machine states. For example, varied cache latencies are not allowed in a timing contract, as cache hits/misses depend on the history of cache accesses. In this paper, we start the timing contract with a high-performance single-core processor with parallel pipelines, perfect caches, and no speculative execution so that the latency of each type of instructions, including loads and stores, can be statically specified.

The timing contract can be then exposed to the compiler to schedule instructions, based on which $S(P,T)$ can be directly computed. Actually, modern optimizing compilers have already exploited the hardware timing information including latencies of various instructions to schedule instructions intelligently for maximizing resource utilization and attaining the best performance. Thus after compilation, not only the number of instructions but also the scheduling (i.e. static clock cycles) of each instruction can be known. Given a processor P and a trace T, the scheduling time of each instruction in T is usually assigned by the compiler based on a certain scope, e.g. a basic block or a superblock, based on which the statically predicted execution time of a trace can be easily calculated, which is simply called **static scheduling time** in this paper. The details of computing *static scheduling time* for the processor we evaluate can be seen in Section V-A.

Given a trace T, although the instructions of the trace are executed in a processor following the scheduling, their actual execution time may vary at runtime due to the performance-enhancing but non-time-predictable architectural features such as branch mis-predictions and cache misses. This is because the actual processor we implement may not have perfect pipelines, perfect branch prediction, and perfect caches etc. As a result, the actual execution time of a trace T on the given processor P is simply called **dynamic execution time** in this paper, which can be directly measured on a real processor or a cycle-accurate simulator.

Thus given any processor P and any trace T, by applying Equation 1, ATF can be simply calculated in Equation 2. Typically, ATF should be no less than 1[1]. If *architectural time-predictability factor* is 1, it means the architecture is 100% time-predictable. Otherwise, the closer the ATF is to 1, the more time-predictable the architecture is.

$$ATF = \frac{dynamic\ exec\ time}{static\ sched\ time} \tag{2}$$

Why is ATF useful? Researchers in WCET analysis and computer architectures have already figured out certain hardware components such as caches, and branch prediction are not time predictable, so why do we need to use ATF? This is equivalent to say since caches are faster than main memory, a processor with a cache will definitely have better performance than a processor without a cache, thus there is no need to evaluate the actual performance of the processors with or without the cache. When designing a processor, a computer architect usually has multiple design objectives and constraints, including but not limited to average-case performance, energy dissipation, cost, compatibility, and time predictability for real-time systems, etc. It should be noted that while time predictability is surely an important design objective for real-time systems, computer architects are unlikely to only focus on achieving time predictability without considering other important design objectives such as average-case performance. Prior studies on time-predictable design are mostly qualitative in nature, which cannot tell quantitatively how good or how bad the time predictability is, or how much better the time predictability can be improved by applying a new design. With the availability of ATF, it becomes possible to quantitatively study the impact of architectural and microarchitectural design on time predictability, which can be used to make intelligent tradeoffs between time predictability and other design objectives. For example, cache locking is widely known to

[1]ATF may be smaller than 1 in case that we are using a superscalar processor with out-of-order execution so that the dynamic execution sequence leads to less execution time than the **static scheduling time** predicted by the compiler. In this case, the ATF is less than 1, and the smaller the ATF, the more unpredictable the processor is.

provide better time predictability for cache accesses. However, once a piece of data is locked into a particular cache block, that cache block cannot be dynamically reused to hold other data. As a result, the cache performance may degrade. For a processor that needs to balance time predictability and performance, designers might want to only lock a fraction of data or optimally reserve a fraction of cache space for locking while leaving the remaining cache lines for regular caching to achieving higher performance, which can be guided by ATF (for time predictability) and the execution time (for performance).

Is ATF larger than 1 useful? An ATF of 1 indicates perfect time predictability, which is an important design goal of hard real-time and safety-critical systems. However, there could be multiple architectural and microarchitectural designs that can achieve an ATF of 1, but with different impact on performance or energy. Thus, being able to evaluate the ATF of different architectural and microarchitectural design is crucial in this process. By comparison, without the ATF, it would be hard to validate the perfect time predictability, especially for complex processors. Moreover, today soft real-time systems, such as iphones or other handheld devices are widely and increasingly used in our society, for which the quality of service (QoS) is important. Unfortunately, conventional architectural design such as multiprocessor present severe challenges when trying to provide even soft real-time guarantees [11]. Thus, achieving an ATF close to 1, but not necessarily exact 1, could be beneficial for a wide variety of soft real-time systems, for which reducing the time variation, jitters and providing QoS are important.

Note that several prior studies [1], [10] used estimated WCET to compute time predictability. In this paper, we use *static scheduling time* instead of WCET. The estimated WCETs often have different amount of overestimation, which can hardly make the time predictability evaluation accurate. In other words, the inaccuracy of WCET analysis should not be a reason to prevent us from deterministically evaluating time predictability. In contrast, *static scheduling time* is based on the compiler-generated schedule and the timing behavior specified in the timing contract, both of which are deterministic for a given trace. Also, since every program needs to be compiled before execution (the discussion on interpretation and dynamic compilation is out of the scope of this paper), the methodology to estimate *static scheduling time* can be generally applied to different programs and various processors to provide a solid foundation for evaluating architectural time predictability.

IV. Qualitative Analysis of ATP on a VLIW Architecture

In this paper, we validate the effectiveness of ATF on a VLIW architecture based on HPL-PD [12], which is a parametric processor architecture aiming at improving instruction level parallelism (ILP) by adopting advanced compiler and architectural techniques. In a VLIW architecture, the compiler, not the hardware, is responsible for orchestrating the ILP of programs. To facilitate compiler scheduling, the VLIW architecture exposes as much hardware and timing information as possible to the compiler, such as the latency of each instruction, the number of functional units etc. Therefore, a VLIW processor is relatively more time-predictable than a superscalar processor, which dynamically schedules instructions by hardware. However, the HPL-PD based VLIW processor still has some architectural features that can compromise architectural time predictability as the following:

Branch architecture of HPL-PD not only replaces conventional branch instructions with a set of instructions to initiate a prefetch of the branch target early to minimize delays, but also uses a combination of bimodal branch predictor and global history with index sharing to predict the branch target dynamically [13]. In case of a branch mis-prediction of conditional branches, some stall time is added into the execution time at run-time.

Speculative execution in HPL-PD consists of control speculation and data speculation. Control speculation represents code motion across conditional branches. Data speculation is designed to increase the range of code motion for memory instructions. Speculative execution is generally safe but may lead to exceptions. If an exception is raised during the execution of a necessary speculated instruction, the recovery of the exception requires the re-execution of some instructions, resulting in additional execution time. Also as exceptions can only be detected at run-time, speculative execution can possibly degrade architectural time predictability of the processor with the handling and recovery of any exception.

Cache memories of HPL-PD consist of first-level instruction and data caches and a second-level unified cache. Since the latency to access the memory hierarchy for an instruction depends on the result of accessing the caches (i.e. a hit or a miss), which can only be precisely known at run-time, the compiler always optimistically assumes a hit in the first-level cache for each memory access. Thus cache memories can lead to time unpredictability.

V. Evaluation Methodology

We evaluate the ATP of the VLIW architecture based on Trimaran [14], which is an integrated compilation and performance monitoring infrastructure of VLIW architectures. We select 6 real-time benchmarks from Mälardalen WCET benchmark suit [15] and 4 benchmarks from MediaBench [16] for the experiments. The general information of these benchmarks is shown in Table I.

The simulated processor is configured with 2 integer ALUs, 2 float ALUs, 1 branch unit, 1 load/store unit and 2-level caches. The 2-level caches consist of a level-1 instruction cache, a level-1 data cache and a level-2 unified cache. The parameters of the level-1 instruction cache are: size 512 bytes, block size 16 bytes, direct-mapped, miss penalty 7 cycles; the parameters of the level-1 data cache include: size 1024 bytes, block size 32 bytes, direct-mapped, miss penalty 10 cycles; and the parameters of the level-2 unified cache are: size 2048 bytes, block size 64 bytes, direct-mapped, miss penalty 100 cycles. Note that due to the small sizes of the benchmarks,

especially the real-time benchmarks, we use small cache configurations in our evaluation.

In a statically-scheduled VLIW processor, whenever there is a cache miss, the whole instruction pipeline will be stalled to wait until the data is returned. Therefore, the *dynamic execution time* of a trace running on the VLIW processor can be computed based on Equation 3, where $compute\ time$ is the execution cycle through the pipeline, $cache\ stall\ time$ is the stall cycles caused by cache accesses, and $branch\ stall\ time$ is the stall cycles caused by branch mis-predictions.

$$dynamic\ exec\ time = compute\ time + cache\ stall\ time + branch\ stall\ time \quad (3)$$

In order to study not only the architectural time predictability of the processor but also that of each architectural component, we define the following three component-level ATFs to indicate the ATP of speculative execution, caches, and branch prediction respectively. It should be noted that the component-level ATF just studies the effect of an unpredictable microarchitectural component on ATP, thus its value could be less than 1, and 0 indicates that this component does not have negative impact on architectural time predictability.

$$speculative\ ATF = \frac{(compute\ time - static\ sched\ time)}{static\ sched\ time} \quad (4)$$

$$cache\ ATF = \frac{cache\ stall\ time}{static\ sched\ time} \quad (5)$$

$$branch\ predictor\ ATF = \frac{branch\ stall\ time}{static\ sched\ time} \quad (6)$$

A. Static Scheduling Time Analysis

To quantitatively evaluate architectural time predictability of an architecture, static scheduling time of a trace must be analyzed accurately. In contrast, *dynamic execution time* can be easily obtained through simulation or measurement. In the HPL-PD architecture, the main idea of the *static scheduling time* analysis of a trace is to accumulate the *static scheduling time* of all basic blocks (BB)s according to the control flow and the scheduling time determined by intermediate representation(IR) of the program and the given input, which is described in Algorithm 1.

The algorithm begins with determining the weights (i.e. the execution frequencies) of all BBs and its edges in the trace by control flow profiling based on the given input. Then the scheduling time of each instruction in the trace is calculated in the scope of the BB according to pipeline scheduling. Lines 7 to 13 calculate the *static scheduling time* of BBs with exit edges. An exit edge of a BB represents a control flow going out of the BB. In Trimaran framework, a real instruction is executed, while a non-real instruction is not executed. If the source instruction of an exit edge is a real instruction, the *static scheduling time* of one execution of the BB related to this exit edge equals to the scheduling time of this instruction plus 1; otherwise, it only equals to the scheduling time of this instruction. The *static scheduling time* of the executions of the BB from an exit edge equals to the *static scheduling time* of one execution multiplied by the weight of this exit edge. Then the *static scheduling time* of the BB is the sum of the *static scheduling time* of the executions from all exit edges. In case of a BB without any exit edge as shown from Lines 14 to 20, the *static scheduling time* of one execution of the BB is calculated with the scheduling time of its last instruction. Then the *static scheduling time* of the BB equals to the *static scheduling time* of one execution multiplied by the weight of the BB. The algorithm is terminated when the *static scheduling time* of all BBs are accumulated, and its timing complexity is linear to the total number of the exit edges of the trace.

Algorithm 1 *Static Scheduling Time Analysis*

```
1: input: intermediate representation of the program and an input
2: output: static scheduling time of the trace
3: begin
4: Control_Flow_Profiling(IR, input)
5: Pipeline_Scheduling(IR)
6: for all BB do
7:     for each exit_edge of BB do
8:         if src_inst of current exit_edge is a real inst then
9:             BB_time+=(src_inst.sched_time+1) × exit_edge.weight
10:        else if src_inst of current exit_edge is a pseudo inst then
11:            BB_time+=src_inst.sched_time × exit_edge.weight
12:        end if
13:    end for
14:    if no exit_edge in BB then
15:        if last_inst of BB is a real inst then
16:            BB_time=(last_inst.sched_time+1) × BB.weight
17:        else if last_inst of BB is a pseudo inst then
18:            BB_time=last_inst.sched_time × BB.weight
19:        end if
20:    end if
21:    static_sched_time+=BB_time
22: end for
23: return static_sched_time
24: end
```

benchmark	static sched time	dynamic exec time	ATF
crc	20774	20774	1
edn	37655	37655	1
lms	260940	260940	1
matmult	81395	81395	1
ndes	46005	46005	1
statemate	1154	1154	1
cjpeg	12390627	12390627	1
djpeg	3839632	3839632	1
mesamipmap	25787205	25787205	1
mesatexgen	76954216	76954216	1

TABLE II
ATF OF ALL BENCHMARKS IN AN IDEAL VLIW PROCESSOR.

VI. Experimental Results

A. An Ideal VLIW Processor

First, we perform experiments on an ideal VLIW processor, which disables speculative execution and has a perfect cache and a perfect branch predictor. As shown in Table II, architectural time-predictability factors of all benchmarks are exactly 1. These data reveal that architectural time predictability of an ideal VLIW processor is perfect as one would expect.

benchmark	category	description	code size (bytes)	data size (bytes)
crc	real-time	cyclic redundancy check computation on 40 bytes of data	664	458
edn	real-time	finite impulse response (FIR) filter calculations	13504	3104
lms	real-time	lms adaptive signal enhancement	2136	1296
matmult	real-time	matrix multiplication of two 20 × 20 matrices	480	4828
ndes	real-time	complex embedded code	3580	986
statemate	real-time	automatically generated code	2476	498
cjpeg	mediabench	jpeg image compression	71468	135565
djpeg	mediabench	jpeg image decompression	70516	26508
mesamipmap	mediabench	OpenGL graphics clone: using mipmap quadrilateral	124892	39397
mesatexgen	mediabench	OpenGL graphics clone: texture mapping	180228	45074

TABLE I
GENERAL INFORMATION OF ALL BENCHMARKS

B. A Realistic VLIW Processor

Figure 1 demonstrates ATFs of all benchmarks for a realistic VLIW processor. The bar of each benchmark in this figure consists of four components, including the normalized *static scheduling time*, speculative ATF, cache ATF and branch predictor ATF. The ATFs range from 1.26 to 11.18, and are 4.67 on average, indicating the realistic VLIW architecture is not fully time-predictable, which is consistent with our qualitative analysis in Section IV. We notice that the benchmark `statemate` has the worst ATF value. This is because it is a small benchmark that only takes 1154 computation cycles, so the memory stall time due to cache misses (mostly cold misses) becomes significantly larger than the *static scheduling time*, leading to a very high ATF value.

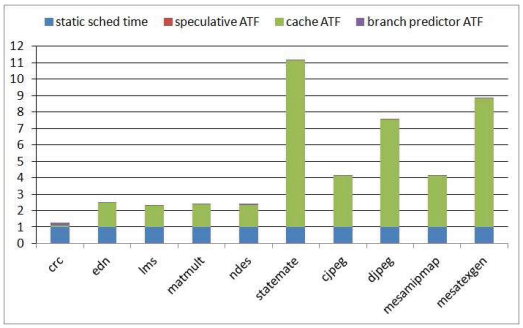


Fig. 1. ATFs of all benchmarks in a realistic VLIW processor.

Table III gives the speculative ATFs, cache ATFs and branch predictor ATFs of the realistic VLIW processor. We observe that speculative ATFs are 0 for all benchmarks except *mesamipmap* and *mesatexgen*. This is due to the fact that only these two benchmarks have both instructions executed speculatively and the exceptions caught as shown in Table IV. On average, the speculative ATFs are still near 0, implying that while speculative execution can affect ATP, its impact is actually negligible for the VLIW processor we studied.

Table III also shows that branch predictor ATFs of all benchmarks range from 0.0078 to 0.1835, and are 0.0449 on average. The time variation between *static scheduling time* and *dynamic execution time* is due to the time of flushing the pipelines in case that the instructions on the mis-predicted paths are executed before the branch targets are known.

benchmark	speculated inst	exceptions
crc	0	0
edn	0	0
lms	0	0
matmult	0	0
ndes	0	0
statemate	8	0
cjpeg	9462	0
djpeg	7588	0
mesamipmap	599172	10
mesatexgen	824499	10

TABLE IV
THE NUMBER OF SPECULATED INSTRUCTIONS AND EXCEPTIONS.

benchmark	branch inst	branch stall time	mis-prediction
crc	5553	3812	953
edn	4121	600	150
lms	28537	4956	1239
matmult	9707	1912	478
ndes	6209	3724	931
statemate	59	52	13
cjpeg	2160542	551320	137830
djpeg	197424	55784	13946
mesamipmap	3563318	201828	50457
mesatexgen	6787772	1072312	268078

TABLE V
THE NUMBER OF BRANCH INSTRUCTIONS AND THE BRANCH MIS-PREDICTIONS OF ALL BENCHMARKS

Although the combined branch predictor is used in the VLIW processor, branch mis-predictions still occur and lead to the stall time. As shown in Table V, branch stall time of each benchmark is proportional to the number of mis-predictions, which means ATP of the branch predictor can be improved by increasing the accuracy of the branch prediction. However, branch prediction only degrades the ATP of the processor by a comparatively small degree, because branch stall time is a relatively insignificant portion of the total *dynamic execution time*.

Additionally, cache ATFs of all benchmarks range from 0.0779 to 10.1334, and are 3.627 on average as shown in Table III, which means time variation from memory hierarchy is not predictable. Because cache ATF is about 77% of ATF for all benchmarks on average, architectural time predictability of the VLIW architecture in study is mostly affected by time predictability of memory hierarchy.

benchmark	static sched time	compute time	cache stall time	branch stall time	speculative ATF	cache ATF	branch predictor ATF
crc	20774	20774	1619	3812	0	0.0779	0.1835
edn	37655	37655	54973	600	0	1.4599	0.0159
lms	260940	260940	336656	4956	0	1.2902	0.019
matmult	81395	81395	111573	1912	0	1.3708	0.0235
ndes	46005	46005	61850	3724	0	1.3444	0.0809
statemate	1154	1154	11694	52	0	10.1334	0.0451
cjpeg	12390627	12390627	38510460	551320	0	3.108	0.0445
djpeg	3839632	3839632	25169447	55784	0	6.5552	0.0145
mesamipmap	25787205	25787375	79860330	201828	0.00000659	3.0969	0.0078
mesatexgen	76954216	76954331	602816266	1072312	0.00000149	7.8334	0.0139

TABLE III
SPECULATIVE ATFS, CACHE ATFS AND BRANCH PREDICTOR ATFS OF A REALISTIC VLIW PROCESSOR.

C. Impact of The Number of Integer ALUs

The number of ALUs in a processor is another important factor that can affect ILP and the average-case performance. However, its impact on time predictability is not clear. Since the arithmetic instructions of the benchmarks in our experiments are mainly integer instructions, we perform some sensitive experiments on the number of integer ALUs (IALUs), which ranges from 1, 2 to 4.

As expected, increasing the number of IALUs reduces the dynamic execution cycles of each benchmark as shown in Table VI. However, it does not imply that the time predictability will also become better. Actually as shown in Figure 2, ATF of each benchmark is increased with more integer ALUs, indicating worse time predictability. This is because with a larger number of IALUs, the compiler can also schedule more operations per cycle, leading to less *static scheduling time*. Interestingly, we found the reduction of *static scheduling time* is more than the *dynamic execution time*. The reason is that in a realistic HPL-PD processor, cache misses or branch misprediction can have greater impact on performance with more operations scheduled per cycle. However, this does not mean that changing the number of IALUs is inherently not time-predictable.

To verify our hypothesis mentioned above, we also conduct experiments with 1, 2 and 4 IALUs on the ideal VLIW processor. We find that the ATF is always 1 regardless of the number of IALUs and the *dynamic execution time* is reduced with the increase of IALUs. Therefore, changing the number of IALUs (or generally the functional units) itself should not affect the time predictability; however, due to its interaction with other time-unpredictable architectural components such as caches and branch predictors, the architectural time predictability of the processor could be affected.

D. Scratchpad Memory

Scratchpad memories (SPMs) [17] are used in embedded processors to improve time predictability and power efficiency. In a scratchpad memory system, the mapping of program and data elements is performed either by the user or by the compiler using a suitable algorithm, resulting in predictable memory access time. In order to evaluate the effect of SPMs on ATP, we replace the 2-level caches in the processor with corresponding 2-level SPMs [18] including: a level-1 instruction SPM, a level-1 data SPM and a level-2 unified SPM. The size and the latency of each SPM are the same as the corresponding cache described in Section V.

benchmark	1 I-ALU	2 I-ALU	4 I-ALU
crc	33480	26205	26000
edn	101131	93228	89507
lms	605272	602552	591197
matmult	197997	194880	185282
ndes	125812	111579	108881
statemate	13076	12900	12512
cjpeg	53068587	51452407	50467769
djpeg	33063023	29064863	28794668
mesamipmap	113238593	105849533	103897385
mesatexgen	685328738	680842909	676586630

TABLE VI
THE *dynamic execution time* WITH THE NUMBER OF INTEGER ALUS VARYING FROM 1, 2 TO 4.

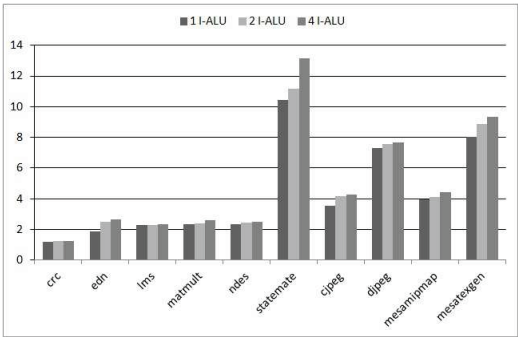


Fig. 2. The ATF with the number of integer ALUs ranging from 1, 2 to 4.

In our SPM allocation method, both instructions and data of a trace are assigned to SPMs by the compiler in the descending order of the number of accesses until all SPMs are filled. The assignment starts from the level-1 SPMs. For the level-2 unified SPM, a fair assignment policy is adopted for simplicity, that is a half of the level-2 unified SPM is assigned to instructions and data respectively. The same policy based on the number of accesses is used for the level-2 SPM allocation as well.

As shown in Figure 3, ATFs of the processor with SPMs are much less than those of the processor with caches, indicating using SPMs can significantly enhance architectural time predictability. On average ATF of the processor with SPMs is

benchmark	cache	spm	spm/cache ratio
crc	26205	24600	93.88%
edn	93228	795655	853.45%
lms	602552	611683	101.52%
matmult	194880	1050607	539.10%
ndes	111579	313057	280.57%
statemate	12900	9618	74.56%
cjpeg	51452407	510420442	992.02%
djpeg	29064863	225820936	776.96%
mesamipmap	105849533	680353365	642.76%
mesatexgen	680842909	7072252472	1038.75%

TABLE VII
DYNAMIC EXECUTION TIMES OF ALL BENCHMARKS IN A PROCESSOR WITH SPMS COMPARED WITH THOSE IN A PROCESSOR WITH CACHES.

1.02 and it is 3.65 times less than that of the processor with caches. Because the memory stall time of a trace depends on the assignment of instructions and data on SPMs, it can be calculated precisely after the compilation and included in the *static scheduling time* for the processor with SPMs. However, the ATF of the processor with SPMs is still not 1, which is mainly caused by timing variation due to branch mis-prediction and mis-speculative execution with exceptions.

However, *dynamic execution times* of all benchmarks except `crc` and `statemate` are increased by using SPMs instead of caches, as shown in Table VII. This is because the assignment of instructions and data in SPMs is fixed and no space in SPMs can be used by multiple instructions/data, leading to longer memory stall time in case the total size of instructions and data is larger than the size of SPMs or caches. In contrast, the caches can dynamically reuse the limited space to get better memory performance. For `crc` and `statemate` however, due to their small code and data footprints, all their instructions and data can be totally assigned into SPMs, hence leading to better performance. In summary, compared to caches, SPMs can significantly improve ATP; however, they can possibly degrade the average-case performance of the processor if the SPM space is not used efficiently[2].

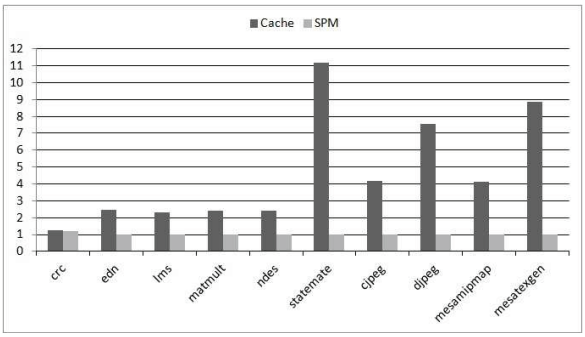


Fig. 3. ATFs of a processor with SPMs compared with ATFs of a processor with caches.

[2]Please note this is just based on the SPM implemented in our experiments, which is not an optimal SPM allocation method. Also, dynamic SPM allocation may improve performance by reusing the SPM space more efficiently; however, this is out of the scope of this paper.

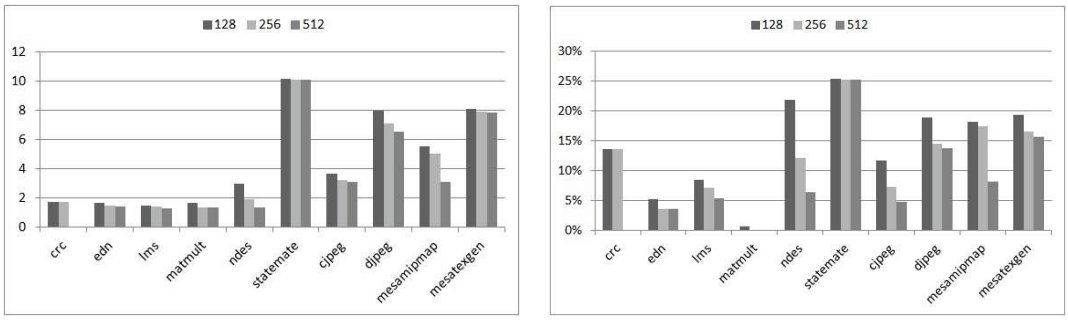
E. Sensitive Experiments of Cache Size

We have also performed sensitivity analysis to examine the impact of different cache sizes on cache ATF. In sensitive experiments of the L1 instruction cache, the size of the L1 instruction cache ranges from 128 bytes, 256 bytes, to 512 bytes; while the size of the L1 data cache is fixed to be 1024 bytes, and the size of the L2 unified cache is fixed to be 2048 bytes (other parameters are the same as those described in Section V). As shown in Figure 4(a), cache ATF of each benchmark except `statemate` is decreased with the increase of the L1 instruction cache size, because cache stall time is reduced with the decrease of the L1 instruction cache miss rates as depicted in Figure 4(b). For `statemate`, it is a very small benchmark whose instruction accesses suffer mostly from cold misses, thus increasing the instruction cache size does not lead to noticeable reduction on the instruction cache misses and *dynamic execution time*. Consequently, the impact on ATF is insignificant.

We also observe that both `crc` and `matmult` have small code size. Thus when the instruction cache size increases to 512 bytes and 256 bytes respectively, their instruction cache miss rates drop to very small values (i.e. 0.12% and 0.3%). The cache ATF of `crc` decreases to 7.8% when the instruction cache size is 512 bytes, because `crc` also has small data footprint and the cache stall cycles are dominated by instruction cache misses. By comparison, `matmult` has larger data footprint, thus its cache ATF decreases when the instruction cache size is increased to 256 bytes but stays almost the same when the instruction cache size is increased to 512 bytes.

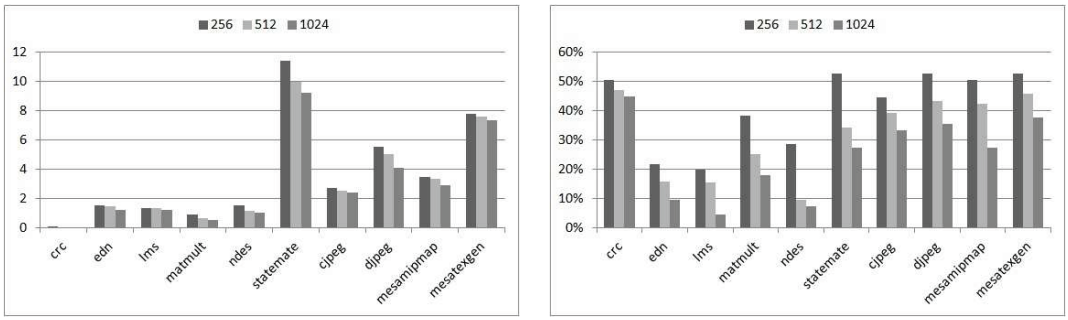
In sensitive experiments of the L1 data cache, the size of L1 data cache ranges from 256 bytes, 512 bytes, to 1024 bytes; the size of L1 instruction cache is always 512 bytes; and the size of L2 unified cache is always 4096 bytes (other parameters are the same as those described in Section V). As demonstrated in Figure 5(a), cache ATF of each benchmark is decreased with the increase of the L1 data cache size, because cache stall time is reduced with the decrease of the L1 data cache miss rate as shown in Figure 5(b). We notice that while `crc` is a small benchmark with small data footprint, most of its data accesses are cold misses, thus increasing the L1 data cache size does not significantly reduce its data cache miss rate. Since the cache stall cycles are only a small fraction of the total execution cycles for `crc`, its ATF is very small as compared to other benchmarks. Specifically, the ATF is 9.3%, 7.5%, and 6.5% when the L1 data cache size is 256 bytes, 512 bytes, and 1024 bytes respectively.

In sensitive experiments of the L2 unified cache, the size of L2 unified cache ranges from 2048 bytes, 4096 bytes, to 8192 bytes; the size of both L1 instruction and data caches are fixed to be 512 bytes (other parameters are the same as those described in Section V). As shown in Figure 6(a), cache ATF of each benchmark is decreased with larger L2 unified cache sizes, because cache stall time is reduced with the decrease of the L2 unified cache miss rate as depicted in



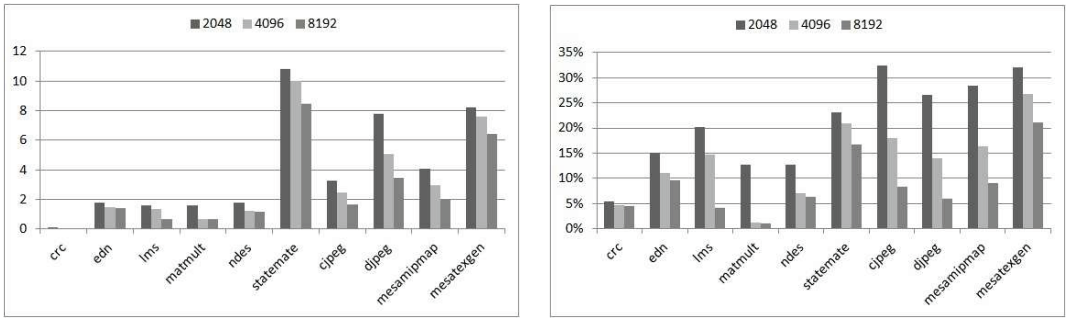
(a) cache ATF (b) L1 instruction cache miss rate

Fig. 4. Cache ATF and L1 instruction cache miss rate sensitive to the size of L1 instruction cache.



(a) cache ATF (b) L1 data cache miss rate

Fig. 5. Cache ATF and L1 data cache miss rate sensitive to the size of L1 data cache.



(a) cache ATF (b) L2 unified cache miss rate

Fig. 6. Cache ATF and L2 unified cache miss rate sensitive to the size of L2 unified cache.

Figure 6(b). Overall we find increasing the L2 cache size is most effective at improving ATF due to its effectiveness on reducing the cache stall time. However, increasing the cache size also adds hardware cost and may increase the cache access latency, therefore there is a trade-off designers should make.

VII. Conclusions

In order to guide the time-predictable architectural design for enhancing time predictability, we present the concept of architectural time predictability to separate the timing uncertainty concern of hardware design from software. Then we propose a new metric named architectural time-predictability factor to quantitatively evaluate architectural time predictability. The availability of such a metric allows computer architects to quantitatively evaluate the impact of different architectural/microarchitectural techniques on time predictability of processors, in addition to other important design objectives such as performance and energy dissipation, thus enabling them to make intelligent tradeoffs among time predictability, performance and energy consumption, which often conflict with each other. Without a metric like this, making quantitative tradeoffs will be impossible, and design for time predictability is at most an art, not science.

Our evaluation on a VLIW processor demonstrates that the proposed metric can effectively assess architectural time predictability of the processor, as well as architectural time predictability of various architectural and microarchitectural

components. More specifically, our evaluation indicates that while speculative execution, branch prediction and cache memories can all affect architectural time predictability, caches have the most significant impact on ATP of the VLIW processor we studied. Moreover, our experiments quantitatively show that using large caches can improve both performance and time predictability; increasing the number of functional units can improve performance but degrade time predictability (though not inherently); and using SPMs instead of caches can increase time predictability but may degrade performance.

This paper is our first step towards quantitatively studying time-predictable and high-performance design of microprocessors. Based on ATF, we can also evaluate the impact on architectural time predictability for a variety of architectural techniques, such as out-of-order execution, prefetching, multi-threaded and multicore execution etc., which can help us either enhance time predictability of existing processors or create new architectures for better time predictability. Moreover, we plan to use ATF to make better and quantitative tradeoffs between time predictability and performance to support both hard and soft real-time computing and/or a mix of real-time and non-real-time applications with different criticalities.

REFERENCES

[1] L. Thiele and R. Wilhelm, "Design for time-predictability," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, ser. Dagstuhl Seminar Proceedings, L. Thiele and R. Wilhelm, Eds., no. 03471. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2004/2

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[3] M. Delvai, W. Huber, P. Puschner, and A. Steininger, "Processor support for temporal predictability - the spear design example," in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, july 2003, pp. 169 – 176.

[4] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555764

[5] N. Yamasaki, I. Magaki, and T. Itou, "Prioritized smt architecture with ipc control method for real-time processing," in *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, april 2007, pp. 12 –21.

[6] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 264–265. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278545

[7] M. Schoeberl, "Time-predictable computer architecture," *EURASIP J. Embedded Syst.*, vol. 2009, pp. 2:1–2:17, January 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/758480

[8] D. Grund, "Towards a formal definition of timing predictability," in *Workshop on Reconciling Performance with Predictability*, Grenoble, France, 2009.

[9] D. Grund, J. Reineke, and R. Wilhelm, "A template for predictability definitions with supporting evidence," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 2011.

[10] R. Kirner and P. Puschner, "Time-predictable computing," in *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, Waidhofen, Austria, 2010.

[11] J. Lee and K. Asanovic, "Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors," *Proc of the 12the IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, 2006.

[12] V. Kathail, M. S. Schlansker, and B. R. Rau, "Hpl-pd architecture specification: Version 1.1," HP Laboratories Palo Alto, Tech. Rep., 2000.

[13] S. McFarling, "Combining branch predictors," Western Research Laboratory, Tech. Rep., 1993.

[14] "Trimaran homepage," http://www.trimaran.org. [Online]. Available: http://www.trimaran.org

[15] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," B. Lisper, Ed. Brussels, Belgium: OCG, Jul. 2010, pp. 137–147.

[16] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," *Microarchitecture, IEEE/ACM International Symposium on*, vol. 0, p. 330, 1997.

[17] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, ser. CODES '02. New York, NY, USA: ACM, 2002, pp. 73–78. [Online]. Available: http://doi.acm.org/10.1145/774789.774805

[18] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," *Design Automation Conference*, vol. 0, p. 628, 2002.