A GENERAL FRAMEWORK TO REALIZE AN ABSTRACT MACHINE AS AN ILP PROCESSOR WITH APPLICATION TO JAVA

WANG HAI CHEN
(*B. Eng. (Hons.), NWPU*)
(*M.Sci., NUS*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgments

My heartfelt gratitude goes to my supervisor, Professor Chung Kwong YUEN, for his insightful guidance and patient encouragement through all my years at NUS. His broad and profound knowledge and his modest and kind personal characters influenced me deeply.

I am deeply grateful to the members of the Computer System Lab, A/P Dr. Weng Fai WONG, and A/P Dr. Yong-Meng TEO, who provided me some good advices and suggestions. In particular, Dr. Weng Fai WONG in later period gave me some good suggestions which are useful to enhance my experiment results.

Appreciation also goes to the School of Computing at National University of Singapore that gave me a chance and provided me the resources for my study and research work. Thanks Soo Yuen Jien for his discussion on some of the stack simulator architecture design work. Also thank the labmates in Computer Systems Lab who gave me a lot of help in my study and life at NUS.

I am very grateful to my beloved wife, who supported and helped me in my study and life and stood by me in difficult times. I would also like to thank my parents, who supported and cared about me from a long distance. Their love is a great power in my life.

Table of Contents

Summary

Abstract machines bridge the gap between a programming language and real machines. This thesis proposes a general purpose tagged execution framework that may be used to construct a processor. The processor may accept code written in any (abstract or real) machine instruction set, and produce tagged machine code after data conflicts are resolved. This requires the construction of a tagging unit, which emulates the sequential execution of the program using tags rather than actual values. The tagged instructions are then sent to an execution engine that maps tags to values as they become available and sends ready-to-execute instructions to arithmetic units. The process of mapping tag to value may be performed using Tomasulo scheme, or a register scheme with the result of instructions going to registers specified by their destination tags, and waiting instructions receiving operands from registers specified by their source tags.

The tagged execution framework is suitable for any instruction architecture from RISC machines to stack machines. In this thesis, we demonstrate a detailed design and implementation with a Java ILP processor using a VLIW execution engine as an example. The processor uses instruction-tagging and stack-folding to generate the tagged register-based instructions. When the tagged instructions are ready, they are bundled depending on data availability (i.e., out of order) to form VLIW-like instruction words and issued in-order. The tag-based mechanism accommodates memory load delays as instructions are scheduled for execution only after operands are available to allow tags to be matched to values with less added complexity. The detailed performance simulations related to cache memory are conducted and the results indict that the tag-based mechanism can mitigate the effects of memory load access delay.

List of Tables

List of Figures

Chapter 1
Introduction

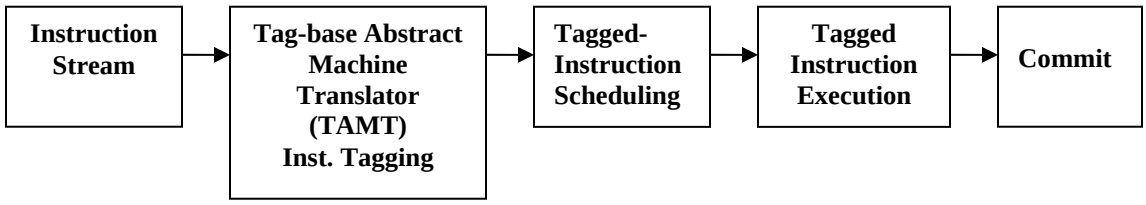
Von Neumann stored-program computers work in instruction-stream driven or control-flow driven style, which is the dominating architecture in modern computer industry [95]. This computer architecture model comprises register-style machines, and stack-style machines. Stack machines [77], which once enjoyed some commercial success (Burroughs 6700, HP3000, ICL2900), are no longer popular among computer architects.

All processors since about 1985 have been using pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* **(ILP)**. A pipeline acts like an assembly line with instructions being processed in phases as they pass down the pipeline. With simple pipelining, only one instruction is initiated into the pipeline at a time, but multiple instructions may be in some phases of execution concurrently. By issuing more than one instruction at a time into multiple pipelines, modern processors are able to achieve high performance with ILP supported.

1.1 Motivation and Objectives

ILP is widely exploited in modern out-of-order processors. An out-of-order processor has the ability to execute instructions by utilizing its ILP potential and identifying dependences among instructions at run time, either through compiling grouping instructions into bundles of non-conflicting members, or through hardware register renaming that resolves data conflicts at execution time. The conventional out-of-order processors in general adopt a superscalar architecture (e.g. PowerPC, Alpha 21264, or MIPS R10000), whereas VLIW (e.g. IA64) processors discover ILP at the compiling stage.

Figure 1.1. The concept of General Tagged Execution Framework (GTEF)



After investigating the architecture of many modern processors, we propose a conceptual framework for designing high performance pipelined processors, which exploits existent instruction-level-parallelism (ILP) execution components, namely superscalar or VLIW execution engines. This conceptual framework (Figure 1) is referred to as ***General Tagged Execution Framework (GTEF),*** which is suited for

multiple computer architectures, whatever register-based or stack-based processors. The proposed framework is characterized by the concept of hardware abstract machine [4] that converts instructions for a particular abstract machine into a general tag-based instruction format.

The introduction of the concept of Abstract Machine makes GTEF scheme cater for multiple computer architectures. Abstract machines are commonly used to provide an intermediate language stage for compilation. They bridge the gap between the high-level of a programming language and the low-level of a real machine. They are abstract because they omit many details of real (hardware) machines [92]. Most common abstract machines are designed to support some underlying structures of a programming language, often using a stack, but it is also possible to define abstract machines with registers or other hardware components. An interpreter or translator is often used to convert abstract machine instructions to actual machine codes, and can be viewed as a kind of abstract machine pre-processor. A processor could be considered a concrete hardware implementation for an abstract machine that requires no pre-processor [92]. This can be a stack machine or a general-purpose RISC register machine.

In GTEF scheme, instructions of the machine are first converted by a predefined hardware pre-processor into tag-based instructions. The pre-processor (or a tagging unit) may be regarded as an “abstract machine” realized in simplified hardware that goes through a “mock execution” – execution with tags rather than values. In the

process of “mock execution”, there is no actual execution which inputs values into arithmetic pipeline to produce output values, and only tags are removed from stack/registers and new tags representing results are put onto stack/registers. The tagging unit processes the instruction stream sequentially, but much faster than actual sequential execution; because it uses tags only, it can keep up with parallel execution that will take place later when tags have been mapped into values.

In GTEF scheme, the tag-based abstract machine translator (TAMT) is a critical component, which converts any abstract or real machine programs into tag-based instructions for ILP execution, including one or more stages preceding the execution stage that can be implemented in either hardware or software. Almost all modern processors have mechanisms to achieve ILP, either through grouping instructions into bundles of non-conflicting members with compiler support, or through the hardware register renaming (tagging) technique that resolves data conflicts at execution time (and register renaming enables out of order execution more effective.)

The hardware renaming/tagging scheme is specifically designed for different CPUs. For multi-issue superscalar machines that employ Tomasulo [85] scheme (e.g. PowerPC, Alpha), a hardware TAMT would be implemented at the tagging and scheduling stage and a superscalar execution engine would be exploited at execution stage; For VLIW machines (e.g. IA64), a similar conversion would be performed with limited scheduling

by hardware at tagging and schedule stages, and a VLIW execution engine to process bundled instructions will be at the instruction execution stage.

The objective of the thesis is to investigate and demonstrate the applicability of the proposed framework. In the thesis we will introduce with GTEF framework, how to design the special-purposed TAMT for different processors including general-purpose register-based processors (RISC or CISC machines) and stack-based processors. In register-based processors, the TAMT will exploit register renaming techniques to implement an instruction mapping from registers to tags, but to fulfill the instruction tagging a “mock” execution technique using tags will be used. In stack machines, the TAMT will simulate the behavior of a virtual stack machine with tags, and translate stack instructions into tag-based RISC-like instructions, then to use existent ILP execution components which may be superscalar or VLIW execution engine to achieve high performance.

For stack machines, a prominent problem was believed to be the presence of a single architectural bottleneck – stack is viewed as a significant performance obstacle in the dynamic extraction of instruction level parallelism (ILP). That is, with instructions taking operands from the top of the stack and leaving results there, stack programs appear to have a high level of data dependency, and with instructions displaying no source and destination register references (even though the source and destination reference are hidden in stack locations), data dependency relations are supposed to be

difficult to analyze. Under GTEF scheme, we proposed a novel bytecode instruction tagging-scheme. The proposed scheme solves the problem of stack bottleneck in stack machines, and in Java processors. In addition, our proposed Java ILP processor is able to extract more ILP in Java programs, and support out-of-order execution.

We demonstrate how the GTEF scheme works on a stack machine by using a Java processor as an example. In the thesis, the GTEF Framework is applied to design the Java processor which adopts a pipelined architecture. It is essential to create a real TAMT in order to implement a Java processor using GTEF scheme. The TAMT to be used is a hardware “abstract” machine that “mock” executes Java bytecodes with assigning each bytecode instruction a tag, and analyzing the data dependency of the instructions to enable hardware scheduling of execution. The design and implementation of the tagging unit and the Java ILP processor will be discussed in Chapter 4 and 5 respectively.

Now we look at how to apply the GTEF scheme extensively. To fulfill a detailed implementation of a processor, some related issues need to be solved. The first is how to attach available data to the tagged instructions. The attachment can be implemented through the use of real registers that correspond to tags, or through a matching mechanism like the Tomasulo machine. The second is how to schedule the executable instructions and send them to arithmetic units. This can be through multiple synchronized pipes like VLIW, or through individually activating them as in Tomasolu

machines from reservation stations [85] next to the arithmetic units. The third is that if the output of load units and arithmetic units are not buffered using real registers with one register per tag, whether there is need for something like a reorder buffer with locations that may be shared by different tagged data at different times, in order to guarantee that the data that become available before instructions are ready to use, have somewhere to go. The fourth is, since a stack machine with operands used once only, how to retain a repeatedly needed value. The solutions to above mentioned issues will be discussed in Chapter 3, 4 and 5.

1.2 Contributions

The thesis has done extensive research on computer architecture and ILP techniques. To explore the applicability of the proposed GTEF scheme, several state-of-the-art out-of-order processors are investigated, such as MIPS R10000 [43], Alpha 21264 [81], and Pentium [24] processor based on x86 architecture. Stack machines have their special features. Since stack is often viewed as the bottleneck to support ILP in stack machines. To solve this problem, we conducted an extensive investigation on stack machine architecture, and using a Java ILP processor as an example. The proposed Java ILP processor exploits a novel stack renaming (or tagging) scheme to overcome the issue of stack bottleneck and be able to expose more ILP within stack programs. In addition, the relevant issues are discussed.

The thesis has the following contributions:

- A novel general processor design framework is proposed. The novelty lies in that it can be used to build a new processor by exploiting existent ILP hardware components and suitable for multiple processor architectures, register-based or stack-based. In this framework, the concept of tag-based abstract machine translator (TAMT) is introduced.
- A stack instruction tagging scheme is proposed to implement stack renaming in stack machines, overcome the stack bottleneck and expose more ILP. After stack instruction tagging, stack dependencies are converted to tag-based data dependencies. One of the advanced ILP techniques – dataflow -- may be exploited to extract ILP in stack programs.
- Stack instruction folding, an efficient technique to reduce stack instruction dependencies in Java processors, is investigated in the thesis. To integrate instruction folding into the proposed Java ILP processor, we proposed a new tag-based POC (Producer-Operator-Consumer) approach which combines POC [50] scheme with stack instruction tagging and can fold almost all bytecode instruction sequence with simple hardware support.
- To apply the GTEF scheme, we designed and implemented a Java ILP processor in which the proposed stack instruction tagging technique is exploited and a VLIW execution engine is used to execute tag-based instructions. Using a VLIW execution engine causes a simpler hardware architecture than using a Superscalar execution engine. Such related issues as instruction schedule, tag management, branch prediction, and speculation support are investigated.

- A trace-driven architectural simulator to model the proposed Java processor architecture was developed. The simulation experiments demonstrate that the proposed Java ILP processor can extract most ILP, and out-of-order execution technique can be exploited to achieve high performance.
- An alternative method called Tag-PFU, to PFU scheme [55] was proposed to tolerate unpredictable memory load delay in VLIW processors. The Tag-PFU scheme realizes the same function as PFU but with tag-based mechanism to accommodate the effects of unpredicted memory load delay. The proposed scheme is more productive and simpler than the previous PFU [55] scheme.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 gives a brief review on abstract machine, ILP techniques, and related works in Java processor and Java technologies including software / hardware scheme, and stack folding, etc. Chapter 3 describes how to apply the GTEF scheme to design new processor architecture by exploiting existing superscalar execution engines, such as Alpha execution engine and Pentium x86 execution engine. Chapter 4 describes how to implement a hardware TAMT in stack machines by using a stack renaming mechanism. Also, a new stack folding scheme is elaborated which combines stack instruction tagging with stack folding technique and a detailed review of stack folding technique is given. Chapter 5 designed and implemented a Java ILP processor by exploiting the TAMT designed in Chapter 4. The

performance evaluation of the Java ILP processor is presented in Chapter 6. Chapter 7 proposes a suspending Instruction buffer (SIB) scheme to solve the memory load delay problem in the proposed Java ILP processor, and cache performance simulation results are given. Chapter 8 gives the concluding remarks of the research work as well as the recommendations for future work.

Chapter 2
Background Review

In this chapter, we will conduct a detailed review of the related techniques to our researches in the thesis, which are abstract machine, ILP, register renaming, etc. We also investigated latest Java-related technologies, e.g. stack folding [28], JIT [1, 6, 15], binary translation [46], multi-threading [82] and some developed Java processors. These techniques have been proposed and implemented by many researchers. After reviewing them, we will get to know a basic research background on microprocessor and Java technology.

2.1 Abstract Machine

Abstract Machines are widely used to implement software compilers. Abstract machines provide an intermediate target language for compilation. First, a compiler generates code for the abstract machine, then this code can be further compiled into real machine code or it can be interpreted. By dividing compilation into two stages, abstract machines increase the portability and maintainability of compilers.

A processor could be considered a concrete hardware realization for an abstract machine that defines the processor's instruction set architecture. This can be a stack machine or a general-purpose RISC processor. From the early 1970s to the late 1980s, since it was believed that efficient implement of symbolic languages would require special-purpose hardware, several special hardware implementation were undertaken [92]. However, with the rapid development of conventional computer hardware, and advances in compiler and program analysis technology, such as special-purpose hardware was no longer to be built due to their very expensive price. Typical such processors are Burroughs B5000 processor – a stack machine architecture, which has hardware support for efficient stack manipulation; the Pascal Micro-engine Computer [103] for the use of UCSD P-code abstract machine; the Transputer [30], a special-purpose microprocessor for the execution of Occam, and some Java processors (picoJava-I, picoJava-II [28, 39]) which directly execute Java bytecode based on Java Virtual Machine, etc. Recently due to its platform independence, compact code size, object-oriented nature and security, Java programming language [104], a static-typed class-based object-oriented language, is widely used from embedded system to high end servers.

2.2 ILP

Instruction-level parallelism (ILP) [22] in the form of pipelining has been around for decades, with systems exploiting ILP dynamically using hardware to locate the parallelism, or using compiler techniques. The amount of parallelism available within a

basic-block is usually quite small. Here a basic block means a contiguous block of instructions, with a single entry point and a single exit point [5]. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

To achieve ILP we must determine which instructions can be executed in parallel, and determine how much parallelism exists in a program and how that parallelism can be exploited. The key point is to see how one instruction depends on another. Thus we need to discuss dependences and data hazards. There are three different types of dependences in a program: data dependences, name dependences, and control dependences. In the following we will discuss them individually.

2.2.1 Data Dependences

An instruction *j* is ***data dependent*** on instruction *i* if either of the following holds:

- Instruction *i* produces a result that may be used by instruction *j*, or
- Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

The first condition states the *data dependence* is a producer-consumer relationship. The second condition simply states that the relationship of data dependence can be recursively constructed a chain of dependences of the first type between the two instructions. And this dependence chain can be as long as the entire program.

To give an example:

```
ADD  R3,  R1,  R2 ; instruction i
ADD  R3,  R3,  R4 ; instruction j
```

As can be seen, instruction i produces the result of addition in register R3, which is used by instruction j. If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. Dependences are a property of programs, and their effect of the dependences must be preserved. This is the read-after-write (RAW) hazard.

The presence of the dependence is a potential limit to the amount of ILP we can exploit. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are dependent on the properties of the pipeline organization. To overcome a data dependence generally has two different ways: maintaining the dependence but avoiding a hazard, and eliminating the dependence by transforming the code. Different computer architectures adopt different techniques. We will discuss the detailed implementation in the later sections.

2.2.2 Name Dependences

A name dependence occurs when two instructions use the same register or memory location (i.e. resource with same name), but there is no flow of data between the instructions associated with that name. In another words, this dependence stems from the utilization conflict of resource, which is partially caused by scarcity of a particular

resource. For example, name dependence may be created when limited number of registers forced the compiler to reuse the same register for an unrelated instruction.

Between an instruction *i* that precedes instruction *j* in program order, there are two possible types of name dependences: *anti-dependence* and *output dependence*.

- When instruction *j* writes a register or memory location that instruction *i* reads, and anti-dependence between instruction *i* and instruction j occurs. In this case, the original ordering must be preserved to ensure that *i* reads the correct value.
- When instruction *i* and instruction *j* writes the same register or memory location, an output dependence occurs. To ensure that the value finally written corresponds to instruction *j* is correct, the ordering between the instructions must be preserved.

Since there is no value being transmitted between the instructions, both anti-dependences and output dependences, are name dependences, as opposed to true data dependences. The name dependence, often called WAR or WAW hazard, is not a true dependence, instructions involved can be executed in parallel or reordered provided that the name (register number or memory location) is changed. The renaming can be easily done for register operands, called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Section 2.3 will discuss the related issues and approaches on register renaming.

2.2.3 Control Dependences

As opposed to the previous two types of dependences, which deal mainly with data values and/or resources, the other type of dependence -- Control Dependences study dependences created by program order (control flow). In brief, the ordering of an instruction is studied with respect to a branch instruction to ensure that execution only occurs for instructions in the correct control path.

The basic rules for control dependence are:

- An instruction i that is control dependent on a branch cannot be moved before the branch. This movement breaks the dependence and allow instruction i to be executed regardless of the outcome of the branch instruction.
- An instruction i that is not control dependent on a branch cannot be moved after the branch. Clearly, this rule is the reverse of the previous one.

Examine the example below (which is written in a C-like syntax):

```
s1;
if (condition){
   s2;
}
```

Moving the statement s1 into the if-block violate the first rule, whereas moving the second statement s2 before or after the if-block violates the second. The rules help to preserve the correctness of the execution by imposing a correct ordering of instructions.

Since most programs are non-linear, which involves multiple control paths, most instructions are under the influence of one branch instruction or the other. If control dependence can be weakened, more instructions will be available for execution. In particular, program loops represents the biggest potential source of speedup.

2.3 Register Renaming

Register renaming is an aggressive way to deal with false data dependences, which assign different physical register names to the multiple definitions of an architected register. Register renaming was first introduced for the float-point unit of the IBM 360/91 by Tomasulo in 1967 [85]. The 360/91 renamed floating-point registers to preserve the logic consistency of the program execution rather than to remove false data dependencies. Nowadays, register renaming becomes a key issue for the performance of out-of-order execution processors and is extensively used.

In out-of-order processors, a typical instruction set architecture may have 32 architected registers while the micro-architecture implements 128 rename physical registers in order to exploit more ILP by simultaneous examining a large window of instructions which have been transformed into a single-assignment language. These rename physical registers contains not only current state but also speculative state (because of speculated branches, loads, etc.)

There are several different register renaming approaches in commercial processors. Here we describe them briefly and the detailed survey can be seen in [20].

The first approach is called the merged register file, in which architectural registers and rename registers are mingled in a single large register file which we call it the physical register file (one for integer and another for FP) to hold both non-committed and committed data. This approach is used in Alpha 21264 [81] and MIPS R1000 [43].

The second approach of register renaming separates rename registers from architectural registers, each have their own register file and are updated appropriately. The non-committed data and committed data are kept in two different register files. This approach is used in PowerPC 603 [94].

The third is similar to the second approach in that non-committed data and committed data are kept in two different register files, but the non-committed data are stored in the reorder buffer (ROB), while copying these data to the register file is needed at commit. This technique is used in the Intel Pentium [24, 51].

Register renaming requires the use of hardware mechanisms at run time to undo the effects of register recycling by reproducing the one-to-one correspondence between registers and values for all the instructions that might be simultaneously in flight. In merged register file approach, it holds that the number of rename registers is greater

than the number of logical registers. This can be simply explained that the rename storage must have enough registers to contain all of the architected state plus some number of registers with speculative state. The other two approaches can completed decouple the rename storage from the logic view of the architecture.

To implement register renaming, a mapping table [84] is often needed to associated limited architectural registers with physical registers in a large physical register file. For example, Intel Pentium 4 exploits a Register Alias Table (RAT), a kind of mapping table, to allow the small, 8-entry, register file architecturally defined in IA-32 to be dynamically expanded to use the 128 physical registers.

2.4 Other Techniques to Increase ILP

Register renaming techniques can reduce data dependences and increase ILP. Besides register renaming, modern high performance processors often exploit multiple-instruction issuing and out-of-order instruction execution technique to improve ILP.

Multi-issue processors are categorized as two basic flavors: superscalar and *VLIW* (very long instruction word) processors. Superscalar processors may issue varying numbers of instruction per clock cycles from zero to the maximum issue rate, and they can be statically scheduled with compiler support or dynamically scheduled with Tomasulo scheme. Statically scheduled processors use in-order execution, while dynamically scheduled processors use out-of-order execution. The early superscalar processors, such

as Sun UltraSPARC II/III adopt static instruction scheduling and recently almost all superscalar processors, such as MIPS R10000 [43], Alpha 21264 [81], PowerPC, and Pentium 4 [24] processor series, use dynamically instruction scheduling.

In contrast to superscalar processors, VLIW processors package multiple operations into one very long instruction word, and the instruction word is inherently statically scheduled by the compiler. VLIW instructions are formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction word. The latter often are known as EPIC – Explicitly Parallel Instruction Computers.

Superscalar processors dynamically can decide how many instructions to issue. A statically scheduled superscalar must check for any dependencies between instructions in the issue packet and between any issue-ready candidates and any instructions already in the pipeline. In order to achieve good performance, it requires significant compiler assistances. However, dynamically scheduled superscalar processors check for any dependencies on the fly with less compiler assistance, but with significant hardware costs.

Alternatively, VLIW processors are to rely on compilers to minimize potential data hazard stalls, as well to actually format instructions in a potential issue packet. To do so, the processor hardware need not check explicitly for dependence. Such an approach

allows VLIW processors to be implemented in simpler hardware through extensive compiler optimization to achieve a good performance.

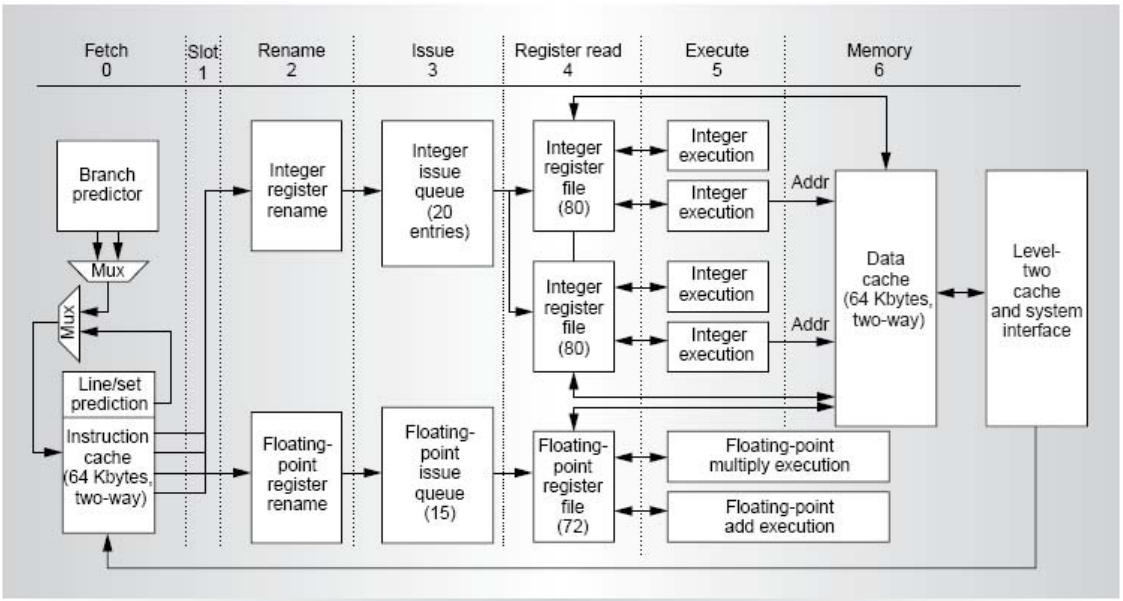
A major limitation of simple pipelining technique is that they all use in-order instruction issue and execution. Instructions are issued in program order, so that if an instruction is stalled in the pipeline, no later instructions can proceed. The idea of dynamical instruction scheduling is to rely on the based hardware to rearrange instructions' execution to reduce stalls while maintaining data flow and exception behavior but come with hardware costs.

Tomasulo scheme eliminates WAR and WAW hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that out-of-order write does not affect any instructions that depend on an earlier value of an operand. Register renaming is often implemented with the use of the reservation stations (RS) and issue logic. RSs can fetch and buffer operands of instructions waiting to issue, eliminating the need to get the operand from a register. Meanwhile, pending instructions designate the RS that will provide their input. Finally, when successive writes to a register overlap in execution, only the last write is actually used to update the register. The use of RSs has two advantages: one is that it distributes hazard detection and execution control, and the other is that execution results are passed directly to functional units from the RSs.

By now, we have reviewed some ILP techniques in modern high performance processors because exploiting ILP is the major technique in processor design to improve processors' performance. Subsequently, we discuss a typical out-of-order superscalar RISC processor -- DEC Alpha 21264 [81] and a VLIW processor – Itanium [29] processor. Its pipeline can be modified to fit for our tag-based GTEF scheme; while our tag-based scheme has features of superscalar processors.

2.5 Alpha 21264 -- a Out-Of-Order Superscalar Processor

Figure 2.1. Stages of the Alpha 21264 instruction pipeline



The Alpha 21264 is a superscalar microprocessor that can fetch and execute up to four instructions per cycle. It also features out-of-order execution and using speculative

execution to maximize performance. The instruction pipeline of the Alpha 21264 (shown in Figure 2.1) has six stages [81]: Fetch, Rename, Issue, Register Read, Execute and Retire.

Instructions are fetched from a 64-Kbyte, two way set-associative instruction cache which offers much-improved level-one hit rates compared to the 8-Kbyte,direct-mapped instruction cache in the Alpha 21164. Four instructions can be delivered to the out-of-order execution engine each cycle.

The 21264 implements a sophisticated tournament branch prediction scheme, which uses two types of branch predictors – local history and global history predictor to predict the direction of a given branch. The tournament branch predictor is a two-level predictor. The first level holds 10 bits of branch pattern history for up to 1024 branches. The global predictor is a 4096-entry table of a 2-bit prediction counters indexed by the path history.

The capability of out-of-order execution contains register renaming, instruction issue logic, and instruction retire logic. The out-of-order execution logic receives four instructions every cycle, renames registers, and queues the instructions until operands or functional units become available. The 21264 can dynamically issues up to six instructions every cycle. It has four integer ALUs, and two float-point units. Although it

issues instructions out-of-order, it provides an in-order execution model via in-order instruction retire.

The issue queue logic in the 21264 maintains two pending instruction lists to separate integer and float-point instructions. As their operands of the pending instructions become available, the queue logic selects from these instructions using register scoreboards. These scoreboards maintain the status of the internal registers by tracking the progress of all kinds of different latency instructions. The dependent ready-instructions can issue as soon as the bypassed result become available from the functional unit or load.

The 21264 fetches and retires instructions in-order. The retire mechanism assigns each mapped instruction a slot in a circular in-flight window (in fetch order). After an instruction starts executing, it can retire whenever all previous instructions have retired. An exception causes all younger instructions in the in-flight window to be squashed, and these instructions are removed from all queues in the system.

2.6 The Itanium Processor – a VLIW/EPIC In-Order Processor

The Itanium processor [29] is the first implementation of the IA-64 architecture which is a VLIW processor. The processor core has the ability of up to six issues per clock, with up to three branches and two memory references. The memory hierarchy consists

of a three-level cache. The first level splits instruction and data caches. The second and third levels are unified caches, and the third level is an off-chip 4MB cache.

The IA-64 architecture introduces the concept of the instruction group, which is a sequence of consecutive instructions with no register data dependences among them. All the instructions in a group could be executed in parallel if there are sufficient hardware resources. Instructions within an instruction group are divided into instruction bundle, which contains three instructions each. The instruction bundles format the fixed instruction formatting. There is a stop bit to differentiate different instruction groups. To simply the decoding and instruction issue process, the template field is used to specify what types of execution unit each instruction in the bundle requires. The ISA architecture designed in this way can achieve implicit parallelism among operations in an instruction and fixed formatting of the operation field, while maintaining greater flexibility than a VLIW normally allows.

The Itanium processor uses a 10-stage pipeline which is divided into four major parts: Front-end, Instruction delivery, Operand delivery and Execution. The Itanium processor can prefetch up to 32 bytes (2 bundles) per clock into a prefetch buffer, which can hold up to 24 instructions. It uses a multilevel adaptive predictor like in P6 micro-architecture. In delivery stage, it distributes up to six instructions to the execution engine. Within this stage, register renaming for both rotation and register stacking are implemented. In operand delivery stage, the following operations will be completed:

accessing the register file, performing register bypassing, accessing and updating a register scoreboard, and checking predicate dependences. The scoreboard is used to detect when an independent instruction can proceed, so that a stall of one instruction in a bundle need not cause the entire bundle to stall. There are nine functional units in the Itanium, two integer units, two memory units, three branch units, and two float-point units, they are all pipelined. In execution stage, it also detects exceptions and posts NaTs, retires instructions and performs write-back.

The high performance of the IA-64 depends on the coordination of compiler and hardware architecture. IA-64 extended the capability of ILP by providing predicate execution semantics. Predicate execution semantics allows compiler to execute instructions from multiple conditional paths at the same time, and to eliminate the branches that could have caused misprediction. Predication is performed in IA-64 by evaluating conditional expressions values in a special set of 1-bit predicate registers. Nearly all instructions can be predicated. The concept of predicate execution provides a very powerful way to increase the ability of an IA-64 processor to exploit parallelism, reduce the performance penalties of branches, and support advanced code motion. Besides that, IA-64 also provides effective register sets to support software pipelining to expose as much as loop-level parallelism as possible.

In the following, we will review some Java and related technologies for increasing the performance of Java execution since our major work involves in the design and implementation of a Java ILP processor.

2.7 Executing Java Programs on Modern Processors

Java [104] is widely used from high end servers to low end hand-held gadgets. Java applications running on high-end server are typically executed using JIT compilers to achieve high performance. In this section we will first discuss the JIT related issues.

However, the memory requirement of JIT compilers is prohibitively expensive for embedded systems and pervasive computing application. So the dedicated Java processors are favored for embedded applications. Java processor adopts a typical stack machine's architecture, thus direct execution of the bytecodes on stack based embedded processors is invariably constrained by the limitations of the stack architecture for accessing operands. In the next section we will discuss related issues of Java processors.

In the following we will discuss them accordingly.

a. JIT – Just-In-Time Execution

Java bytecodes may be executed on various platforms by interpretation or Just-In-Time (JIT) compiling. The first Java virtual machine (VM) available was interpreter-based, but it was neither efficient nor well-suited to high performance applications. The JIT

compiler translates bytecodes to the native code of the host machine dynamically. Several variants of the JIT concept [6, 15] have been proposed.

Unfortunately, the JIT method suffers some drawbacks. They can usually only perform limited optimizations because time for more sophisticated analysis is not available. Furthermore, JIT systems often optimize only selected sections of code, leaving many segments to continue executing in the interpreter. Finally JIT systems are sufficiently large and complex that they incur runtime overhead in translating bytecodes to native codes, although acceptable performance for Java applications can be provided. Especially in embedded field, using JIT compilation causes an unacceptable wait between application launch and an application actually running on an embedded device. Thus dynamic adaptive compilation (DAC) [46] is proposed to overcome these drawbacks of JIT.

b. Dynamic Compilation Techniques

In DAC scheme, Java method classes that are most heavily used are compiled and optimized in traditional compiler technique in order to obtain more efficient native machine code. A DAC combines a JIT compiler and a bytecode interpreter. The heavily used code sections are often identified by a software profiler. When performed statically, a single profiling run is taken to be representative of the program's behavior. Within a dynamic optimization system, the ongoing profiling identifies which part of

codes are currently hot, allowing optimizations to focus only where they will be most effective. However, DAC scheme still has the following problems.

First, an application will run in a slow interpreter mode until code has been profiled, then pause to generate compiled code. When an application is launched, many methods are only run once, so ideally should never be compiled. This impact can be very significant, particularly at application start-up. Second, because software interpretation is very slow, most DAC solutions do very little profiling and compile almost all methods immediately, making guess that a method is not about to be executed for the last time, but will be executed many times. This guess is very costly if it is incorrect.

To overcome the above drawbacks, ARM proposed a scheme of hardware-based dynamic compilation – ARM Jazelle technology, which can directly execute Java instructions on ARM RISC architecture [109]. ARM designers added a new Java instruction set to the classic ARM architecture. The Java ISA is executed in a Java mode, which is entered on a branch. In the Java mode, the CPU executes Java bytecode instructions. Bytecodes are fetched and decoded in two stages. Use of Jazelle technology, the compiler can afford to compile less code and interpret more. Jazelle technology can also be used to improve the speed performance of a DAC compiler by holding off compilation. Jazelle technology improves the performance a lot according to ARM’s white paper [109].

2.8 Increasing Java Processors' Performance

Hardware processors to execute bytecode directly are becoming popular. The designs of Java processors, such as picoJava [28, 39], are mainly based on stack processors, and generally Java VM is used as their instruction set architecture. A major issue in implementing Java processor is the existent limitation of ILP by the stack dependence. Several techniques to overcome the limitation in Java bytecode have been investigated [53, 28, 88, 44].

A. Stack Folding

Stack operation folding is one technique to reduce the limitation by converting a set of bytecodes into a RISC-like register-based instruction [4, 48, 50, 70]. In Sun's picoJava-II processor, simple instruction folding in hardware is done by using pattern matching at decode stage of its pipeline [28, 88], and the stack folding is supported by the stack cache as a register file for parallel access of stack operands to eliminate redundant stack operations. More sophisticated folding techniques, such as nested folding [4, 48, 50, 53, 70], may further reduce stack operand dependence. The more detailed stack instruction folding techniques will be discussed in Chapter 4.

B. Multiple Instruction Issue

Combining multiple in-order issue with stack folding is proposed in ILP [88], which proposes to improve the performance of Sun's picoJava-II processor with in-order,

dual-issue bytecode execution, a fill unit, and stack disambiguation, but this work does not consider out-of-order bytecode execution, which would naturally exploit a greater degree of ILP in Java programs. To support out-of-order execution, SMTI [79] is proposed with software involved to extract independent bytecode trace and implement bytecode folding, but special fetch logic is needed to identify independent traces from instruction cache.

C. Multi-threading

To meet the requirement of high-performance network application with Java, thread level parallelism (TLP) can be exploited to extract coarse-grained parallelism. Sun's MAJC processor adopts a vertical multithreading technique, in which Java methods are treated as a thread in hardware and speculative execution of multiple threads is included to exploit TLP [69]. But MAJC needs a JIT compiler to convert bytecodes to native codes. The Java Multi-Threaded Processor (JMTP) [82] architecture is a similar hardware implementation, which is a single-chip CPU containing an off-the-shelf general purpose processor core coupled with an array of Java Thread Processors (JTPs). However an intelligent compiler is needed to identify the set of concurrent threads that can be forked as JTP threads.

D. Dynamic Translation

DAISY [47] is designed on VLIW architecture with dynamic translation, which combines JIT with native compilation techniques by appropriate hardware primitives designed to execute Java efficiently. It dynamically translates Java bytecodes with JIT

into VLIW instructions and exploits a VLIW engine. This approach can take advantage of the increased ILP possible in VLIW machines to achieve high performance [46].

The Femtojava [8] and Delft-Java [40] are another two dynamic-translation-supported Java processors. The FemtoJava processor is a stack-based architecture with replicated functional units and instruction decoders, and employs a VLIW as its execution engine. In FemtoJava [8], the bytecodes in the entire Java program are divided into the instruction groups, the instructions within the same group are translated into VLIW word to be executed. The grouping algorithm is to find those instructions that depend on the result of the previous one, and group them in one instruction block. The Delft-Java [40] processor provides hardware assisted dynamic translation, and the bytecodes are translated on-the-fly into the Delft-Java instruction set. Hardware support for Java language constructs are incorporated into the processor's ISA. This allows application level parallelism inherent in Java language to be utilized ILP.

E. Some Dedicated Java Processors

Along with the Java widely used in embedded field, some dedicated Java processors are proposed and built. We will introduce them in the following.

Espresso [110]

Aurora VLSI's Espresso Java processor is a superscalar RISC engine. The CPU has two operational units, each with an integer and a floating-point processing unit. Espresso supports a 32-bit 128-entry stack. It has 32 to 256 on-chip registers (configurable) and supports 16k to 32k instruction and data caches with 64-bit interfaces. It executes four instructions/cycle or seven bytecodes/cycle.

Lightfoot Java CPU [111]

Digital Communications Technologies' Lightfoot is a direct-execution Java CPU with a one-to-one mapping between bytecodes and lightfoot instructions. This design tactic eliminates the interpreter and keeps Java's small program memory footprint. The 32-bit Harvard RISC processor provides stack execution for both Java and C. It implements an eight-register-deep stack, with extensions to data memory. The soft core supports J2ME, JavaCard, KVM, and JINI.

JStar [112]

Nazomi Communications' JStar can work with ARM, and MIPS. JStar's Java translation mechanism is automatically invoked whenever the main processor's instruction pointer falls within a specified memory address range. Java code is simply placed in this memory and can be called directly. JStar uses the processor's registers, including the stack registers, to handle calls just like native code.

2.9 PicoJava -- a Real Java Processor

In this section we will discuss a typical stack processor – picoJava [28,39], which directly execute Java bytecode based on a stack processor architecture. The processor uses a pipeline structure to achieve good performance. The Figure 2.2 shows the basic pipeline of the PicoJava-II [28] core.

Figure 2.2. Basic pipeline of the PicoJava-II

FETCH	**DECODE**	**REGISTER**	**EXECUTE**	**CACHE**	**WRITEBACK**
Fetch fixed size cache lines to the I-Buffer	Predecode & group instructions	Access the register file (stack cache) for operands	Execute for one or more cycles	Access the data cache	Write back results into the operand stack, Forward results

PicoJava is a comparable RISC processor architecture. PicoJava core contains the integer execution unit and a compact floating-point unit with separated instruction and data caches, which are 16Kbytes. A Java processor must execute all 226 bytecode instructions defined for the Java virtual machine. The 226 instructions can be divided into 15 different functional categories. To efficient execute Java bytecode, picoJava categories bytecodes into three classes: simple, moderately complicated, or very complicated. The simple instructions are RISC-like in the sense that they readily lend themselves to hardware implementation. These instructions are hardwired and execute in a single clock cycle. The majority of instructions executed by a typical program would fall into this category, such as all the integer arithmetic operations. The group of

moderately complicated instructions contains about 30 bytecode instructions, and they are implemented using microcode. Microcode offers a good balance between the need to keep the hardware implementation simple and the need for good performance. The last group of about 30 instructions are either very complicated or require services from the underlying operating systems. They can be executed by a software emulation trap.

A stack processor must spend cycles moving operands to the top of stack in order that the compute operations can get at them, and moving results off the top of stack for storage. These stack manipulation operations makes stack processors pay an overhead burden of up to 30 percent more than RISC processors. To reduce this overhead, PicoJava adopts a register file with 64 entries to support stack operations. The register file treats as a circular buffer, with a pointer to the top of stack. The register file has three read and two write ports. Compute operations can simultaneously read out two operands and write back one result. All data from the constant pool, from local variables, or loaded from objects, are first pushed onto the stack and all compute instructions then access their operands from the stack, and push the results back onto the stack.

The register file, functioned as stack cache, provides a powerful solution to the problem of access inefficiency in stack machines. This leads to an execution technique called instruction folding. The instruction folding can fold up to four bytecode instructions into a RISC-like register-based instruction, by taking the operation to be performed from the compute instruction, the source of the operands from the local variable loads,

and the destination of the results from the local variable store. The instruction folding eliminates essentially all of the computational overhead of stack processors, achieving the same sort of single-cycle execution efficiency found in RISC processor architecture.

Chapter 3

Implementing Tag-based Abstract Machine Translator in Register-based Processors

Chapter 1 proposed a concept of General Tagged Execution Framework (GTEF). The tag-based abstract machine translator (TAMT) is a critical component for GTEF scheme, which converts any abstract or real machine programs into tag-based instructions for ILP execution. The concept of TAMT is similar like a software-supported dynamic instruction translator, and the one of merits of TAMT is that it can support dynamic instruction translation and easily collaborate with existing RISC / CISC processors. For example, after designing a TAMT to dynamically translate RISC instruction into tag-based instruction formats, we can only design and implement different TAMT to translate different register-based ISA into a common tag-based instruction formats. Then we can use existing ILP execution engine, PowerPC or Pentium execution engine to tag-based instructions with micro-code support.

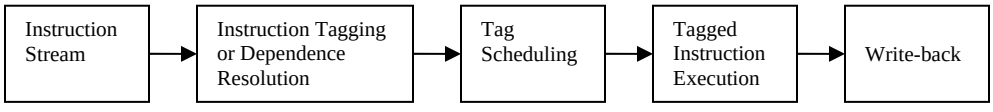
In this Chapter, we first discuss a general design methodology of TAMT for register-based processor architecture, and then with several general-purpose register-based

processors – Alpha 21264 and Pentium processor as examples to discuss how to design different TAMT in order to use them as the execution engine.

3.1 Design a TAMT

The key-point of the proposed GTEF scheme is to convert any machine program (abstract or real) into tag-based instructions. Conceptually, the instruction execution procedure in GTEF can be described as in Figure 3.1. The conceptual framework captures many existing computer architectures, since one or more stages preceding the execution stage can be implemented either in hardware or software. For superscalar (multi-issue) machines that employ Tomasulo scheme (e.g. PowerPC, Alpha), the tagging and scheduling stage would be implemented by reorder buffer and common data bus in hardware and the execution stage would be utilizing a superscalar execution engine; For VLIW machines (e.g. IA64), the tagging and schedule stage would be performed in software (the compiler) while with limited scheduling in hardware and the fourth stage will be a VLIW execution engine to process bundled instructions.

Figure 3.1. A conceptual tagged execution framework



In the conceptual framework, a TAMT is built in stage two which is responsible for instruction tagging and/or dependence resolution, and after this stage any inputted instructions are translated into a tag-based instruction format which can be employed by modern ILP execution engines, superscalar or VLIW processors.

TAMT executes with tags rather than values. In TAMT, there is no actual execution which inputs values into arithmetic pipeline to produce output values; only tags are removed from stack/registers and new tags representing results are put onto stack/registers. The TAMT processes the instruction stream much faster than sequential execution. i.e., it can keep up with parallel execution that will take place later when tags have been mapped into values. That's why we name it as "tag-based abstract machine translator". Based on the functional descriptions on TAMT, we know that in addition to playing tag renaming, TAMT is also responsible for dependence resolution among instructions, and instruction scheduling. Thus, in general, a TAMT may consist of a tagging unit (TU), a tag matching unit (TMU), and a free tag pool (FTP). TU can be a virtual tag execution unit, which is a core unit in TAMT. TMU can be responsible for dependence checking and tagged instruction scheduling. FTP is used to store free tags.

In the following we use an example to illustrate the instruction tagging – "mock execution" scheme of implementing with RISC machines.

The TU adds tags to register numbers in order to distinctly represent changing register contents. Given the expression e = a*b+(c+d), the compiler produces the following RISC instructions:

ld *r1,a; ld r2,b; mul r1,r1,r2; ld r3,c; ld r4,d; add r3,r3,r4; add r1,r1,r3 ; st r1, e*

Each register is renamed by attaching an additional tag (which recycles 0,1,2,3 etc). The renaming and "mock execution" process can be seen in Table 3.1. Now observe that the "mock execution" taking place in the tag-renaming unit. For any instruction that modifies a register, the tag-renaming unit merely attaches a new tag to the instruction that will later produce a value, or attaches the same tag to the register being modified, or attaches the same tag later to any instruction that will need the value of the instruction. In actual execution an instruction may be delayed by cache miss, long arithmetic computation or busy units etc, holding back progress, whereas the mock execution proceeds much faster. However, every different value is represented by a tag and an instruction that produces a value can be recognized by the one that consumes it as related.

Table 3.1. A sample of RISC instructions renaming process

Instructions	Tag Renaming Unit	Actual Execution (Later; assume superscalar)
ld r1, A	Ld r1-1,A	ld to load unit; tag 1 added to r1 to denote new value
ld r2,B	ld r2-1,B	ld goes to load unit, tag 1 is added to r2
mul r1,r1,r2	mul r1-2,r1-1,r2-1	goes to reservation station of multiplier – r1 reused so gets new tag
ld r3, C	ld r3-1,C	goes to load unit
ld r4, D	ld r4-1,D	goes to load unit
add r3,r3,r4	add r3-2,r3-1,r4-1	goes to adder
add r1,r1,r3	add r1-3,r1-2,r3-2	goes to reservation station of adder
st r1, E	st r1-3,E	waiting at store unit and executes when previous finishes

Table 3.2. The tag-based RISC-like instruction format

Opcode	Src1 Tag	Src2 Tag	Dest Tag

The above tag-based renaming scheme may work with a processor's pipeline architecture, e.g. implementing between decoding unit and issue logic. The tagging unit collaborated with decoding unit to assign each instruction a tag, which will store the result the instruction executes. With instructions' "mock execution" with tags, a new tag-based instruction format (Table 3.2) is generated and stored in Tag-Matching-Unit (TMU). TMU may be a Reorder Buffer (ROB) structure with attaching a mapping table, which indexed from tag number to physical registers. In the mean time the dependent information among instructions within TMU is generated with tags. The tag-based instruction formats are very similar as that used in most RISC processors, thus it will be easily employed by existing RISC processors. The instruction tagging scheme contains a mapping from old ISA with old architectural registers to a new tag-based ISA with tag number, and another mapping from the tag number to physical registers. For example, tags correspond to individual registers. A value loaded from memory or computed by an arithmetic instruction is retained in a tag / registers till it is de-allocated. In a stack machine, however, a tag for a value loaded onto the stack may be de-allocated after use, as we shall see later.

To make our tag-based scheme easy to be understood, in the following we will illustrate how to implement the TAMT in two conventional RISC processors -- DEC Alphas 21264 [81] and Intel Pentium [51]. Both of processors use different register reaming techniques. We will explain individually.

3.2 Design a TAMT Using Alpha Engine

Table 3.3 illustrates the process of the tag-based instruction renaming in 21264. In this sample, we assume the number of tag registers is greater than that of architectural registers in order that more instructions can be tagged in the instruction window. For load and store instructions, the related data must be read / written to the register first in order that they can be operated continually. Each Alpha instruction is assigned a tag, at the same time register renaming is implemented. Here we follow the Alpha instruction definition, put the destination field at the right most. The register renaming in 21264 has separate integer and float-point renaming unit. A unique tagging unit may be implemented in the architecture, and it executes instructions “mock” with tags. Tag is un-typed, so which can pointer to both integer and float-point values.

Table 3.3. A sample of tag-based renaming for Alpha processor

Instructions	TAMT Renaming Unit	Tag-based Instructions Generated

ldl $1, (A)
ldl $2, (B)
mull $1,$1,$2
 	
ldl $3, (C)
ldl $4, (D)
addl $3,$3, $4
addl $1,$1,$3
stl $1, (E)
	ldl $1-1, (A)
	ldl $2-2, (B)
	mull $1-3, $1-1, $2-2
	
	ldl $3-4, (C)
	ldl $4 -5, (D)
	addl $3-6, $3-4, $4-5
	addl $1-7, $1-3, $3-6
	stl $1-7, (E)
		ldl T1, (A)
		ldl T2, (B)
		mull T1, T2, T3
		
		ldl T3, (C)
		ldl T4, (D)
		addl T4, T5, T6
		addl T3, T6, T7
		stl T7, (E)

From the Table 3.3, we can see that the Alpha instructions are dynamically translated into the tag-based RISC instruction format, and this format is similar as the RISC instruction format for Alpha processors, so it is easy to be integrated with the previous design. The previous used superscalar execution engine can be continually employed. Here we can see that our TAM scheme is easy to be applied with the existent out-of-order execution engine.

3.3 Design a TAMT Using Pentium Engine

X86 instruction set is a CISC instruction set with variable instruction length. To execute x86 at high performance, Intel's Pentium [24, 51] dynamically translates x86 instructions into simple, fixed-length instructions that Intel calls micro-operations or uops. These uops are then executed in a decoupled superscalar core capable of register renaming and out-of-order execution. Like RISC instructions, uops use a load / store

mode [51]. Those x86 instructions operating on memory must be broken into a load uop, an ALU uop, and possibly a store uop [51]. Uops use a regular structure to encode an operation, two sources, and a destination like RISC instruction.

Table 3.4 shows how to apply our proposed instruction tagging scheme to convert uops after x86 instruction translation process into the tag-based RISC-like instruction format. Here the same sample program in Table 3.1 is used. We run *gcc* to get the assembly code in first column, after instruction translation, we get the tag-based code in the column 4. The code is similar like in Table 3.3, and the instruction format follows the definition in Table 3.2.

Table 3.4. A sample of tag-based renaming for Pentium processor

Instructions	**Convert to UOPs**	**TAMT Renaming Unit**	**Tag-based Instructions Generated**

Movl (a) , %eax
movl %eax, %ecx
imul (b), %ecx
 	
movl (d), %edx
movl (c), %eax
addl %edx, %eax
addl %ecx, %eax
movl %eax, (e)
	load (a), %eax
	mov %eax, %ecx
	load temp, (b)
	imul temp, %ecx
	load (d), %edx
	load (c), %eax
	add %edx, %eax
	add %ecx, %eax
	store %eax, (e)
		load (a), %eax-1
		mov %eax-1, %ecx-2,
		load temp-3, (b)
		imul temp-3, %ecx-4,%ecx-5
		load (d), %edx-6
		load (c), %eax-7
		add %edx-6, %eax-7, %eax-8
		add %ecx-5,%eax-8,%eax-9
		store %eax-9, (e)
			load (a), T1
			mov T1, T2
			load T3, (b)
			imul T3, T4, T5
			load (d), T6
			load (c), T7
			add T6, T7, T8
			add T5, T8, T9
			store T9, (e)

The above tag-based instruction format may be easily used by Pentium execution engine through combining the register renaming logic, since the instruction format is followed the common RISC instruction format. The register renaming logic in Pentium

renames the logical IA-32 registers onto the processors 128-entry physical register file. A Register Alia Table (RAT) is used to remember the mapping relationship. Our tag renaming unit may contain up to 128 entries which equals to the number of physical register in order to meet the Pentium's performance requirement.

We have demonstrated our tag-based scheme how to be integrated with existing architecture in order to use existent ILP execution hardware. With collaborated with individual register renaming logic, the tag-based scheme may translate different instruction format into RISC-like format, then modern superscalar execution engine can be exploited, so reusability of these existent superscalar component is extended.

3.4 Discussion on Implementation Issues

To implement a real TAMT in order to collaborate with existing RISC execution engines, we can exploit the existing register renaming mechanism provided in the processors. A common way to implement register renaming in RISC processors is to use a separate rename register file (RRF) and the architected register file (ARF). A simple way to implement the RRF is to simply duplicate the ARF and use the RRF as a shadow version of the ARF. Most modern processors implement RRF with much more entries than that of ARF in order to increase instruction-level parallelism. However, this does require a mapping table which gives for each name the index of the physical register with which the name is currently associated in ARF. A common used register

renaming scheme which uses a separate RRF in conjunction with a mapping table to perform renaming of the ARF is illustrated in Figure 3.2.

Figure 3.2. Common register renaming scheme in RISC processors

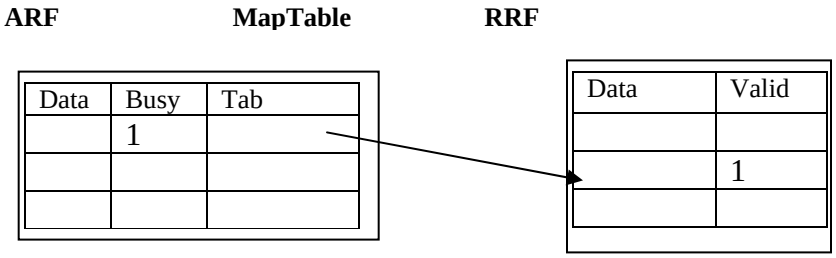
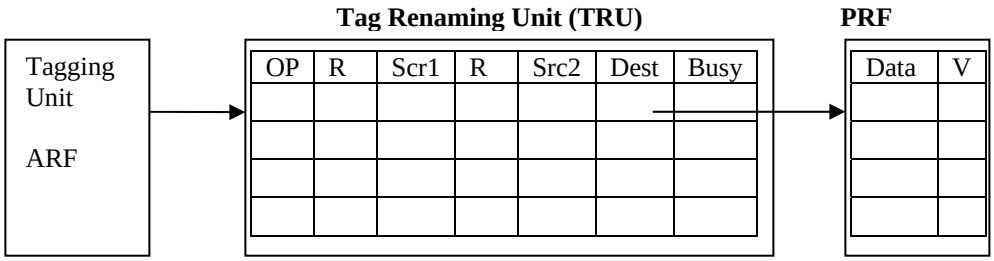


Figure 3.3. Tag-based renaming mechanism



The TAMT can be implemented between decoding unit and issue logic. Figure 3.3 illustrated a common scheme to implement a TAMT. In this scheme, the tagging unit (TU) works with decoding unit to assign each instruction a tag. With instructions' "mock execution" with tags, a new tag-based instruction format is generated and stored in Tag Renaming Unit (TRU). TRU is organized as a Reorder Buffer (ROB), and instruction's dependent relationship is also established with tags. A mapping table is attached to TRU, which indexed from tag number to physical registers.

By means of this dynamic translation process, a newly build processor which uses our proposed approach can use existing RISC execution engines -- superscalar or VLIW as a component. This will reduce the complexity of CPU design.

Based on above discussion, we will discuss some architecture issues when using Pentium [51] and Alpha 21264 [81] execution engines. The further investigation will be our future work.

3.4.1 Implementing Issues using Alpha Engine

To implement a TAMT is not complex with Alpha Engine. A tagging unit (TU) is needed to execute instructions “mock” with tags. The mapping table of TRU can index to both integer register file and float-point register file. The TRU can be organized as ROB to commit instructions in order. Design in this way, TAMT can manage speculative states which is consistent with the Alpha engine’s architecture.

After instruction tagging, the out-of-order execution engine can be employed without need to make any changes. The instruction issue queue and register status updating are similar, no need to change.

3.4.2 Implementing Issues Using Pentium Engine

The register renaming logic in Pentium renames the logical IA-32 registers onto the processors 128-entry physical register file, which is organized as a Reorder Buffer

(ROB) [51]. A Register Alia Table (RAT) is used to remember the mapping relationship, which remembers the most current version of each register, such as EAX so that a new instruction coming down the pipeline can find the correct place to get the correct current instance of each of its input operand registers. The ROB entry has data field and status field. The ROB data field is used to store the data result value of the *uop*, and the ROB status field is used to track the status of the *uop* as it is executing in the machine. These ROB entries are allocated and/or deallocated sequentially. Upon retirement, the result data in ROB data field is physically copied into the separate Retirement Register File (RRF).

To implement TAMT on Pentium, the TAMT may be between decoder and register renaming stage. Since the IA-32 decoder decodes X86 instruction into *uops* in program order, TAMT can follow with the decoder and collaborate with decoder to tag *uops* into tag-based instruction formats which are stored in TRU to implement the function of re-ordering, tracking, and sequencing instructions and help manage physical register file. The TAMT can hold as many as 128 entries to reach the size of physical register file of Pentium. To reduce the complexity, each tag entry would point to one entry in physical register file. The TRU may be integrated with the RAT and commit tagged instructions in order. After above change, the newly generated instructions can still use out-of-order Pentium engine.

Chapter 4

Realizing a Tag-based Abstract Machine Translator in Stack Machines

This chapter will discuss how to apply our proposed GTEF scheme to implement a stack processor. The stack processor has its specialty – stack is often viewed as a bottleneck to achieve the performance. In this chapter a stack instruction tagging (SIT) scheme will be presented to overcome the stack bottleneck problem. The SIT scheme is the groundwork for implementing the tag-based abstract machine translator (TAMT) in stack machines.

4.1 Introduction

Stack programs appear to have a high level of data dependency with instructions taking operands from the top of the stack and leaving results there, and due to with instructions displaying no source and destination register references, data dependency are thought to be difficult to analyze. The SIT scheme proposed in this Chapter can overcome the problem of the stack bottleneck in stack machines by converting stack dependency into

tag (register) dependency. With instruction tagging, the independent stack instruction groups with stack dependences are identified. Because there is not existed stack dependences among the different instruction groups, they can be issued and executed in parallel and thus the scheme can extract more ILP. In the following, we first review some other schemes about stack renaming techniques, and then describe the SIT scheme.

4.2 Stack Renaming Review

In previous research, there are two stack renaming techniques worthy of be noted, one is BLP [44], and another is the method proposed in Kapoor [83]. BLP [44] is a software-implemented interpreter, which bears a great deal of similarity to micro-architectural simulators like SimpleScalar [23]. BLP interpreter exploits the virtual register scheme, which contains bytecodes queue, a control unit, stack renaming unit, branch prediction unit and execution queue. The control unit is responsible for mapping stack locations to virtual register using the stack renaming unit. This step is crucial to uncover more bytecode-level parallelism.

In BLP, the stack renaming unit keeps track of the next available virtual destination register. Since no virtual destination register is written twice, this eliminates register-related WAW and WAR hazards. The stack renaming unit also maintains a stack of virtual registers (called renaming stack) that stores the sources of bytecodes not yet processed. The renaming stack would mimic the operations of a real stack to pop or push in order to get the virtual register. For example, a JVM *iadd* operation might be

translated to *vr2 <- iadd (vr1, vr0)*. The renaming stack would be popped twice to get the virtual source registers, vr1 and vr0. The destination register – vr2, would be pushed onto the renaming stack.

The control unit controls the fetching of bytecode, and it can change the flow of control of the program according to the branch prediction unit. The control unit tracks the program order of bytecodes, and commits or squashes their results as appropriate once the outcome of preceding branches has been verified. A SMT processor is suggested to be needed in BLP scheme as an ideal hardware platform to run the BLP interpreter in order to achieve the desired performance.

The another stack renaming scheme is proposed in Kapoor [83], which maps the operand stack to hardware registers, and stack instructions are converted to register instructions naively by associating stack locations with a particular register. A Stack Translation Table (STT) is used to map the stack locations onto the register file. The STT is a stack which stores the register identifiers. In hardware implementation, the STT can be a series of multiplexer that select the register names that provide the operands for each instruction. The associated push/pop for each instruction are encoded in a ROM. Whenever a set of instructions come in, the ROM can be looked up to yield a set selection signals which can be applied to the current state of the STT and the register free-list to yield the register tags for operands. The ROM can be small and fast lookup

can be possible. An alternative solution was suggested to use combinational logic to compute the selection signals.

In addition to the STT, a pool of free registers must be maintained. The free register pool consists of a list of the register identifiers of free registers. Stack renaming and register allocation was done on a value basis, rather than on a 32-bit word basis. Because the renaming is value-based, the proposed processor will contain four register files; one each for integers, long integers, floating point numbers, and double-precision floating point numbers. When there are no free registers available in free register pool, the renaming will stall, thus a register spilling is needed to provide. To implement this scheme a multi-issue superscalar processor is proposed. Compared with them, our "mock" execution tagging scheme streamlines the process of stack renaming, and it can be extended to RISC or CISC processors

4.3 Proposed Stack Renaming Scheme

Subsequently we will show how to reveal the instruction level parallelism (ILP) in stack programs through stack renaming. Because in stack processors those operands on execution stacks are erased once they are used by an operator, an operand only needs to be supplied to one operator which can be uniquely identified by a tag. Once a tag is used, its new result is immediately discarded without being actually stored into the stack; in contrast with general purpose register processors, new register contents must be written back to physical registers from the reorder buffer (ROB) even if they may

already have been superseded by later writes. This scheme exploits a stack of tags rather than a stack of values. In the following we will explain our stack renaming scheme with an example.

Consider the following expression with its corresponding stack machine code:

g = a*b+(c+d)

- LD A, LD B, MUL, LD C, LD D, ADD, ADD, ST G

If above stack codes run on a Superscalar processor, some ILP can be explored. If the fetching of A or B is slow (e.g., cache miss), CPU would proceed with the fetch of C and D into other registers and produce the result of C and D out of order, then dispatch the multiplication instruction for execution as soon as A and B emerges from the load unit and forward the result of A * B to the instruction. The execution behavior can be obtained after executing the following stack renaming procedure.

Table 4.1. A sample of stack renaming scheme

Instruction	Stack Naming Unit	Operand Tag stack (OTS)
1 load a	T1 load a	T1
2 load b	T2 load b	T1 T2
3 mul	T3 mul T1 T2	T3
4 load c	T4 load c	T3 T4
5 load d	T5 load d	T3 T4 T5
6 add	T6 add T4 T5	T3 T6
7 add	T7 add T3 T6	T7
8 store e	T8 store T7 e	

Table 4.1 shows the stack renaming procedure with renaming stack location with tags and using Operand Tag Stack (OTS) to identify source operands with operators that consume them. The stack renaming unit uses a new tag for every instruction that leaves a result on the stack instead of an operand value. The tags on OTS are used for attachment to a later instruction that consumes the operand.

The procedure of stack renaming shows how the parallelism can be achieved. The tagged instructions are dispatched after the both operands it needs are ready, and the operands may be provided by an instruction which is executed in a load/store or ALU unit and its result is delivered to the later instruction that carries its result. In the snippet program, the first two loads deliver their operands to T3 tag (multiplication operator), and the last two loads to T6 tag (addition operator), then they will be executed and results to be delivered to T7 tag (second add operator), as the same manner in superscalar machines. As we can see that using a stack of tags makes it easy to attach operand tags to an operator. After instruction tagging, the relationship of instruction dependency is established and independent instruction groups may be identified if a group of tagged instructions are not dependent on the other instructions' results.

The proposed stack renaming scheme is data-driven. The tags are organized as a physical register file that can be reused and dynamically assigned to the later coming instructions after they are retired. The single tag entry is composed of the instruction op-code, status bits and a tag sequence number which points to the address of

"destined" physical register. Once an operator instruction is tagged, it will identify its operands by tags. The tag can be seen as a data token as in tagged dataflow machines [9, 11] where the flow of data token activates instructions' execution. In the process of instruction tagging, a data dependence graph (DDG) is generated dynamically and instruction execution may follow the graph. As in dataflow machines, the availability of tagged operands of an instruction triggers its execution and the tagged result as data tokens is passed directly between instructions. The instruction tagging scheme supports explicit out-of-order instruction execution.

4.4 Implementation Framework

Table 4.2. A sample of stack renaming scheme with tag-based instructions

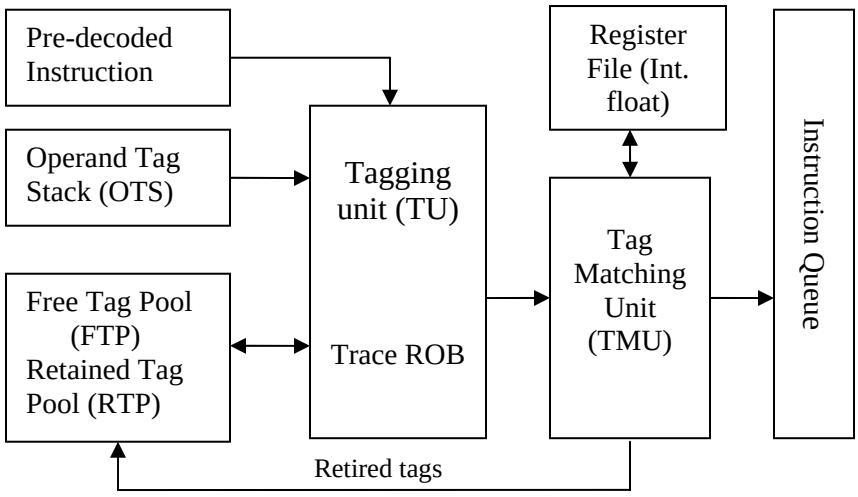
Instruction	Stack Naming Unit	Operand Tag stack (OTS)	Newly Generated Tag-based instructions
1 load a	T1 load a	T1	load T1, (a)
2 load b	T2 load b	T1 T2	load T2, (b)
3 mul	T3 mul T1 T2	T3	mul T3, T1,T2
4 load c	T4 load c	T3 T4	load T4, (c)
5 load d	T5 load d	T3 T4 T5	load T5, (d)
6 add	T6 add T4 T5	T3 T6	add T6, T4, T5
7 add	T7 add T3 T6	T7	add T7, T3, T6
8 store e	T8 store T7 e		store T8, T7

With instruction tagging scheme, the newly generated tag-based instructions are listed in the Column 4 of Table 4.2. We can see that the scheme can translate stack-based instruction into tag-based RISC-like instruction format, and then this instruction format can be easily used in modern RISC processors.

Based on the instruction tagging scheme, we give the logic framework for the TAMT is shown in Figure 4.1. The real TAMT consists of four components: the Tagging Unit (TU), Operand Tag Stack (OTS), Tag Matching Unit (TMU) and Free Tag Pool (FTP). TU is a control unit, which is responsible for tag allocation and release. OTS is an execution engine with tags. If the instruction pops values from the stack, the tag number of the associated operands are popped from the OTS, and entered into the TMU to build the tag-based RISC-like instructions. If an instruction both pushes and pops from the operand stack, all pops occur before any of the pushes occur. FTP may be a resource unit, which provides physical resources to TU for its consuming, and withdraws the released tags.

The stack instruction code is streamlined as following. Whenever a pre-decoded stack instruction enters the TU, TU allocates a tag from FTP. After tagging, the instruction information is stored in TMU. For each tag entry in TMU, the address of the physical register exists in it, which shows the mapping relationship between tag number and physical register. After instruction tagging, the newly generated tag-based RISC-like instructions are inserted into the RISC instruction queue for later scheduling. TU is a simple hardware abstract machine, which “mock” executes stack instructions with tags in program order with OTS support. A retained tag pool is needed with FTP to be used for implementing tag retention and tag reuse. The tag entry in TU can be released by TMU when a tag is no longer used later.

Figure 4.1. Architectural diagram for stack tagging scheme



The detailed stack renaming algorithm is as follows. Like traditional register renaming, our stack renaming approach seeks to avoid false dependencies. When a instruction enter the queue, a free tag entry, which holds the related information, such as tag number, left / right operands, status, is allocated from the FTP, if the instruction is ALU-related a physical register is also allocated to hold the result. The related information is stored in TMU. And TMU keeps this tag and maintains the information until all the readers of the tag have been retired.

In the TAMT, FTP is maintained as follows. FTP stores a list of free tags. A free tag is one which is neither currently assigned to computation nor store any values for later use. When an instruction is retired, the corresponding tag can be returned to the FTP. Each

instruction decoded under goes the tagging process. If the instruction pushes a value on the stack, a tag is allocated from the FTP to hold this value, and the tag number in TMU is set as valid, the value is stored in the entry of TMU.

After stack renaming the all dependency information are generated and stored in TMU, these tag dependency information will be used by instruction issue logic to control tag-based RISC-like instruction issue. TMU can work as a scoreboard to issue instructions out-of-order even though the tag renaming process is in a sequential manner.

4.4.1 Tag Reuse

In stack processors, in addition to stack operands that are used once only, some repeatedly used data are buffered in the register file. They correspond to reused tags. As described previously, a free tag list is maintained for allocation and reuse of tags in FTP. If a tag has been used or its associative value has been read, because it is no longer needed, and can be put back into the FTP for later use by other instructions. In our design, each tag is associated a counter. It is incremented each time the tag is referenced and is decremented each time the associated instruction is issued or an actual read operation is done to get the value from the tag. The tag can be freed whenever the count becomes zero and at this time it can be reused by other instructions.

Note that some instructions like swap and pop, will only affect the OTS and not involve actual execution. In our algorithm a dup instruction merely duplicates the tag number on

the top of the OTS. In the mean time, the reference counter of the duplicated tag is incremented. For swap instruction, only the locations on the OTS of the two tag numbers are exchanged.

4.4.2 Tag Spilling

Although instructions may execute out-of-order, they commit in program order to guarantee the correctness of the program execution. To do this, a Trace Re-order Buffer (TRB) is provided in Tagging Unit (TU) to queue up instruction tags in program order to help implement branch prediction, and precise interrupts.

Tag spilling is needed when there are no free tags available to continue tag the instructions. To do this, some current tags will be removed from the TU, and be copied to the memory. Later re-copy them back from the memory to TU, to execute them. But this will bring some hardware complexity and a spilling algorithm as in RISC processors is needed to provide. To make the system hardware simple, we do not provide spilling function in current design. But we let the processor stop instruction fetching and make TU stalled when TMU is full.

4.5 Hardware Complexity

We have assumed that a certain number of instructions, such as four, can be renamed in every cycle. This section briefly looks at how this might be realized in hardware. Here we only focus on how to implement OTS (Operand Tag Stack) and TMU (Tag

Matching Unit), the other parts could employ the modern existing processor hardware to build.

OTS is responsible for simulating a stack behavior. Thus a hardware stack structure should be provided. It can be implemented by using a linear addressed register file to store tag numbers, and a stack pointer is needed for stack addressing. With Java as an example, we categorized Java bytecodes into seven classes based on the number of stack operands that they pop and push. An example classification follows:

1. No stack movement needed: nop, iinc, goto, ret …
2. 1 Pop needed: pop, istore, ifeq …
3. 1 Push needed: iload, sipush, ldc, icoust_0 …
4. 1 Pop 1 Push needed: ineg, arraylength, i2f, …
5. 2 Pop 1 Push needed: iadd, iaload, fmul, …
6. 3 Pop needed: iastore, fastore, lastore, …
7. Others

With this classification, the stack movement on OTS will follow a map table which is stored in a ROM for fast lookup. With OTS, the dependency information is built, and stored in TMU for later use. After the results returned from the execution engine, the corresponding tag numbers will be directly sent to TMU as direct forwarding in superscalar processors. The lookup and matching of tags are similar as in superscalar processors.

4.6 Stack Folding with Instruction Tagging

The instruction tagging scheme can collaborate with stack folding to further reduce data movements and remove data dependence between stack instructions in stack processors. In this section, we will use Java processor as example to present how to combine both of two functions to increase performance. Subsequently, we first discuss stack folding techniques in Java processors.

Java processors can entirely bypass the need for dynamic translation and reestablish a simple, direct execution model for Java code. But, there is a need to overcome the limitations of the stack architecture for accessing operands. Stack folding [28,39,48] is such a technique to coalesce multiple stack based instructions to a single RISC-style instruction with optimized data accessing. In the following, we will review some previous research work related to stack folding, and propose a new stack folding scheme which exploits instruction tagging scheme.

4.6.1 Introduction to Instruction Folding

The performance of a stack machine has been limited by the true data dependency. This required a performance enhancement mechanism. Such a mechanism called *instruction folding* was first proposed and implemented by Sun Microsystems in their PicoJava [28] processor. It has been seen that this instruction folding mechanism was able to fold up to 60% of the instructions.

In PicoJava, stack cache provides a powerful solution to the classic problem of access inefficiency in stack machines. Since the stack cache is implemented in a full random access register file, the PicoJava pipeline has immediate access not merely to the top two entries on the stack, but to all 64 entries held in the stack cache. This leads the way to the instruction folding technique.

As we know, in stack machine an add operation will need 4 stack instructions (ILOAD_1, ILOAD_2, IADD, ISTORE_3). The two values to be added are likely both already in the stack cache (in the parameters and local variables area of the current method). The problem is that neither happens to be at top of stack. Consequently, the execution unit must spend a cycle moving each operand to the top of stack. Likewise, the local store instruction does not move the returned sum out of the stack, but merely relocates it from the top back into the local variables section of the current method frame. As long as all of these local movements happen within the top 64 elements of the stack—true in an overwhelming majority of the cases for local motion in the stack—they will occur inside the stack cache. Thus it is possible to combine (or fold) these several serial operations together into a single RISC-style add operation.

In general, PicoJava operates on bytecode instructions based on a set of grouping rules, it scans the incoming bytecode streams looking for sequences of instructions that can be folded together (combined into a single operation). These sequences can consist of up to

four bytecode instructions. They are moves of local data to top of stack which are immediately followed by compute instructions that consume the data just moved, and computing operations which are immediately followed by local stores of the result just computed.

When the stack folding core finds such a sequence of instructions, it synthesizes a register-based RISC-style operation, by taking the operation to be performed from the compute instruction, the source of the operands from the local variable loads, and the destination of the result from the local variable store. We can use another example to demonstrate the advantage of folding technology.

In Java Processor, some local variables (LVs) are stored in some register files. The two mathematical expression, c=a-b, and f=d+e, are translated into Java bytecode sequence shown in Table 4.2. Although there is no true data dependence between the two statements, the second expression can not be evaluated concurrent with or preceding the first one as both expressions are using the stack as an intermediate target. Operations have to be issued and executed in the sequence, this results in eight clock cycles to these two expressions.

Table 4.3. Bytecode folding example

Issuing Sequence	Without Folding	With Folding
1	iload_2	iload_2, iload_3, isub, istore_1
2	iload_3	iload_5, iload 6, iadd, istore 4
3	Isub	
4	istore_1	
5	iload 5	
6	iload 6	
7	Iadd	
8	istore 4	

In Table 4.3, Java bytecodes (groups) are issued in sequence starting with the topmost line. Assume both unfolded and folded instructions take one cycle to execute. The middle column shows bytecodes issued one at a time (without folding) consuming a total of eight cycles. The last column shows bytecode sequence issued in groups consuming two cycles. The third column shows how the folding could reduce the number of clock cycles. With folding technology, the two expressions take only two cycles provided that sufficient resources (i.e. load/store units, data paths, etc) are available.

Even if instruction-folding is done to exploit the random access provided by the stack cache and to reduce movement of data for the most common groups of instructions, unfortunately, the permissible groups of instructions that can be folded in this manner are limited in number and scope. In addition, not all redundant data moves are avoided. Although redundant moves between instructions in a group are avoided, there is still

forwarding of data through the registers in the stack cache between groups. To solve this problem, some researchers still proposed the better algorithm to improve this technology, and a higher percentage of the folding rate is obtained. In the following, we will describe some other instruction folding schemes.

4.6.2 Stack Folding Review

JVM [104] is a Java run-time execution environment running on stack machine architecture. In a direct JVM hardware stack implementation, stack access consumes extra clock cycles. Furthermore, individual operations executing on the operand stack one at a time causes data dependency that limits ILP. After instruction folding, multiple-stack-based instructions can be coalesced to a single RISC-style instruction. This not only eliminates some of the data dependency but also allows multiple-instruction issuing and execution.

Although instruction folding is first proposed by Sun, several new methods are proposed. Typically, the major stack folding techniques can be categorized as: pattern matching [28], POC-based (Producer-Operator-Consumer) [53] which includes original POC [53], advanced POC [4], and EPOC (enhanced POC) [50], and Operand Extraction-based (OPE) method [70]. We will describe them accordingly.

Pattern Matching

Table 4.4. Instruction types in picoJava

Types	Descriptions
LV	A local variable load or load from global register or push constant
OP	An operation that uses the top two entries of stack and that produces a one-word result
BG2	An operation that uses the top two entries of stack and breaks the group
BG1	An operation that uses only the topmost entry of stack and breaks the group
MEM	A local variable store, global register store, and memory load
NF	A non-foldable instruction

PicoJava uses pattern matching to implement instruction folding, which can reduce 60% stack operations [28]. In this technique, bytecodes are categorized into 6 types (Table 4.4). Folding logic is added to the decoder to detect the patterns of foldable instruction group. Although there are innumerable folding patterns, only those which occur with high frequency are checked for. Pattern detection is as follows. Since up to four instructions are decoded in the decoder, first only those foldable patterns consisting of four instructions are checked for. If no pattern is detected, check for 3-instruction patterns. If no 3-instruction pattern is detected in this time, check for 2-instruction patterns. If a pattern is detected, the instructions are folded together and one RISC-style instruction is constructed for that pattern.

POC-based Scheme

The basic POC scheme is a pattern matching–based scheme too, but it categorizes bytecode according to their role in stack folding into: *Producer*, *Consumer and Operator* [48]. In this scheme, stack operations like const/load/store are the target bytecode instructions for folding. According to the definition of stack operations folding, the off-chip memory load/store operations cannot be folded, because they will occupy the execution unit for memory access.

Table 4.5. Instruction types in POC method

Roles in Folding	**Types**	**Descriptions**
Producer	L	load from LV/Push Constant
Consumer	S	Store to LV
Operator	O_E	Execution Unit Instructions
	O_B	Branch/Control Transfer
	O_C	Complex/Micro-ROM

The *Producer* instructions push data from on-chip local variable memory or constant registers onto operand stack in a single cycle. The *Consumer* instructions pop data from operand stack and store the data into on-chip local variable memory. The *Operator* instructions pop data from operand stack, execute some kind of operation, then push the result back to operand stack. The instruction folding occurs when some *Producer* instructions produce the data and one or more *Consumer* instruction consume it, or some *Producer* instructions produce the data and the data is processed by one *Operator*

instruction, then the result is written back to stack or consumed by one or more Consumer instructions.

POC scheme divides bytecodes as 5 types (Table 4.5), and gives 2-folable pattern 5 types – LS, LO_E, $LO_{B,}$ $LO_{C,}$ O_ES, 3-foldable pattern 4 types – LLO_E, $LLO_{B,}$ $LLO_{C,}$ LO_ES, and 4-foldable patterns one-type -- LLO_ES. Like in picoJava, POC employs foldable pattern match to implement instruction folding, and it can reduce up to 84% of all stack operations [48]. In this scheme, simulation results reveal that the 3-foldable strategy has the best cost/performance ratio if a size-byte decoder width is provided.

Advanced POC (APOC) Scheme

APOC [4] scheme is a new POC model by extending POC model. It separates O type instructions in POC scheme into another two types: Producible Operator (Op) and Consumable Operator (Oc). Results of bytecode operations always become either producible or consumable types. The APOC model instruction types and their distributions in various applications are shown in Table 4.6. The APOC still uses pattern matching to implement stack folding. All the patterns detected and their occurrences are shown in Table 4.7.

Table 4.6. Advanced POC instruction types

Types	Definitions	Examples	%
P	Producers	iconst_1, iload_3	59.5%
O_p	Producible Operators	iadd,fcmpl	22.0%
O_c	Consumable Operators	if_icmpeq, if_acmpne	4.1%
C	Consumers	iastore,istore_0	14.4%

Table 4.7. Instruction folding patterns and occurrences in APOC

Patterns	Percentage	Patterns	Percentage
P-C	31.7%	P-P-O_P-O_C	0.6%
P-O_p-C	1.0%	P-P-P-C	10.7%
P-P-C	3.6%	P-P-P-O_P-C	8.4%
P-P-O_C	18.9%	P-P-P-O_P-O_C	2.6%
P-O_P-O_p -C	0.6%	P-O_P-P-O_P-C	0.1%
P-P-O_P-C	21.2%	P-P-P-P-O_P-C	0.5%

In APOC scheme, 87% to 93% of foldable instructions are found across the benchmark application programs [4]. Unlike the traditional models, by detecting and folding a broken sequence, the APOC model-based folder is able to find more foldable instructions than the traditional folding mechanisms. In the proposed hardware implementation of APOC, the instruction decoding logic and the fold-ability checker are mapped to generic gates, which can decode up to six contiguous bytecode instructions to determine the fold-ability.

Enhanced POC (EPOC) Scheme

Later the same group of researchers enhances their previous POC model to propose an enhanced POC method [50], which adds a small-sized stack reorder buffer (SROB) to help folding generation. Unlike picoJava, this method does not depend on pattern match. In theory, it can complete almost all folding generation with small-sized SROB, and work in in-order instruction issue mode.

Operand Extraction-based (OPE) Scheme

Table 4.8. Instruction types in OPE algorithm

Types	Symbol	Description
Producer	P	loads stack with a Constant or LV
Consumer	C	stores stack top into a LV
Operator	O	an ALU instruction
Independent	I	increases a LV with a Constant
Destroyer	D	pops stack entries
Duplicator	U	duplicate stack entries
Swapper	W	swaps 2 stack entries
Load	L	loads/allocate an object element
Store	S	stores an element in an object
Branch	B	Branches
Complex	M	a complex operations

The operand-extraction-based (OPE) method was proposed in [70]. In this scheme, bytecodes are categorized into 11 types by their functions (seen Table 4.8) and a finite-state automaton (FSA) is provided to help instruction folding. This method is characterized by dynamic allocating some address on LV stack to store temporary variables, which are intermediate variables generated in folding process as register

renaming process. In its implementation model, bytecode instructions can be issued out-of-order.

4.7 Implementing Tag-based Stack Folding

In the thesis we proposed an instruction folding scheme which exploits instruction tagging mechanism, named tag-POC. It can fold almost all the possible combinations in any Java bytecode sequences without defining instruction folding patterns as EPOC [50]. The tag-POC scheme is designed to fold continuous or discontinuous bytecode sequences with a special hardware – Operand Tag Stack (OTS) support. OTS can store all the tag number of bytecode instructions that have not been folded.

To describe our scheme clearly, we need to look at the mechanism of stack instruction tagging. When an instruction is decoded, a tag number is assigned to it at the same time. This tag number corresponds to the address of a tag entry. This tag entry will store the information of the instruction. Additionally, as described previously, an Operand Tag Stack (OTS) is used to simulate the stack. With OTS, the instruction dependency information is able to be acquired. The following example (Table 4.9) shows the process of generating instruction dependency information in decoding stage.

The basic concept of tag-POC instruction folding model can be observed from above instruction decoding. Here, we give a simplified version of POC types. The Producer instructions push data from local variable onto operand stack. The Consumer

instructions pop data from operand stack. The Operator instructions pop data from operand stack, execute some kind of operation, then push the result back to operand stack. In the tag-POC folding scheme, we categorize bytecode instructions as follows (seen in Table 4.10).

Table 4.9. A sample for dependence information generation

Step	Tag No.	Bytecode	Dependency Tag No.	POC type	Operand Tag Stack status
0	T0	iload_2		P	{T0 }
1	T1	iconst_2		P	{T0, T1}
2	T2	iload 5		P	{T0,T1,T2}
3	T3	iadd	T1, T2	O	{T0,T3}
4	T4	imul	T0, T3	C	{T4 }
5	T5	istore 6	T4	P	{ }

Table 4.10. Instruction type for POC folding model

Roles in Folding	**Type**	**Descriptions**	**sample Instructions**	**Percentage (%)**
Producer	P	Load from LV/push constant to operand stack	iconst_1, iload_2	41.3
Consumer	C	pops the value from operand stack and stores to LV	istore_0	7.0
Operator	O_E	executed in execution unit	iadd, imul	33.4
	O_B	Branch/Control Transfer	if_icmp, goto	7.9
	O_R	executed in micro-ROM	Ireturn	8.3
	O_M	Miscellaneous stack operations	dup, swap	2.1

In order to fold all foldable bytecodes, the tag-POC scheme considers not only POC types of the bytecode stream, but also together with the produced intermediate items. If a bytecode sequence is ILAOD_2, ICONST_2, ILOAD_5, IADD, IMUL, ISTORE_6,

the corresponding POC types are P, P, P, O, O, and C. The sequence of P, P, and O are folded and the execution result (named IP) will be stored in the corresponding tagging register entry – T3. Then folding unit finds that P, IP, O, and C can be folded together. Figure 4.2 shows the process.

Figure 4.2. A sample of tag-POC instruction folding model

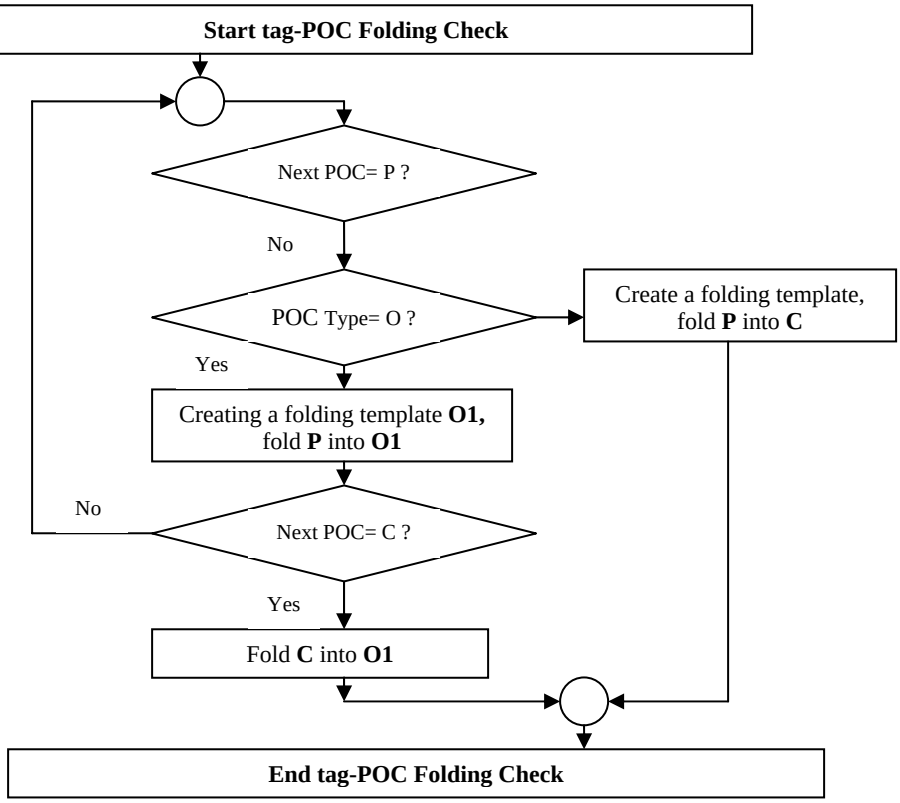
Bytecode stream: **P, P, P, O, O, C**
Step1: P, **{P, P, O},** O, C
Step2: **{P, IP, O, C}**
Finish.

To observe above folding process, OTS can be viewed as an Abstract Stack Machine, which possesses the following features:

1. Based on POC categorization, only three types exists in OTS.
2. Only P or O type bytecodes can produce tag on the stack and Only O or C type bytecodes can consume tags from the stack.

Since the tag number is corresponding to the address of a tag entry, after instruction folding to build a RISC-like instruction becomes easy. With the OTS, a basic stack behavior is simulated and dependencies among instructions are identified. The processing of the tag-POC folding model is shown in Figure 4.3.

Figure 4.3. The process of tag-POC instruction folding scheme



Because the bytecode data dependence analysis is done before the stack folding, in the tag-POC model, we will employ it. We create six stack folding templates including *PO*, *PPO*, *PPPO*, *POC*, *PPOC*, *and PC*. (*OC* type is often combined into *PC* type.) Whenever the folding check logic finds an O type bytecode, it will first create a template, such as PO, PPO, PPPO, according to its instruction characteristics, then check next bytecode. If it is a C type instruction, it can be further folded. Otherwise, the

current folding check process ends and starts next folding check. With stack folding, the tag-based RISC-like instructions are generated accordingly.

It is worth noting that we subdivided the Operator as $O_{E,}$ $O_{B,}$ $O_{R,}$ and O_M in Table 4.9. The $O_{E,}$ $O_{B,}$ O_R instructions need to use execution unit, branch instructions and micro-ROM instruction. But the O_M instructions need special processing. To execute them, (such as SWAP, DUP, DUP2, POP, POP2, etc), we only need to change status of Operand Tag Stack (OTS). For example, POP instruction only causes top element of the stack is removed, DUP instruction will duplicate the top element in the stack. It will not join the instruction folding process, and not change status of virtual registers. What's more, O_R instructions like INVOKE (INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL) instructions are complex operation instructions, which need to be issued individually.

Compared with the previous methods, our proposed algorithm has the following advantages. First it combines POC [50] and OPE [70] method. The POC issues bytecodes in-order and using simple hardware without ILP support while OPE method can issue bytecodes out-of-order but need a FSA structure to implement instruction folding. Our method is different from POC in that it has different instruction categorization, and that stack management instructions are specially processed, which only affect Operand Tag Stack (OTS) and some fields of tag register entries. Since tag-POC scheme can fold bytecode instructions across stack management instructions, the better folding efficiency and performance improvement are obtained. In addition, our

method is different from OPE in terms of register renaming. To issue instruction out-of-order, OPE needed to add temporary variables to local variable (LV) area, then renamed them. This will increase hardware complexity.

4.8 Performance of Tag-based POC Scheme

4.8.1 Experiments Setup

We developed a trace-driven simulator to analyze and evaluate performance of our Java ILP processor. Trace-driven simulation uses a predetermined instruction sequence, and the instruction trace to evaluate microprocessor performance.

Table 4.11. Description of the benchmark programs

Benchmark	Description	count ($x10^6$)
jess	A popular NASA's CLIPS expert system shell	9
db	Data management software from IBM	2
javac	Sun JDK Java compiler 1.0.2	8
mpegaudio	software decompress an MPEG layer 3 audio stream	116
jack	A Java parser generator from Sun	90
Compress	A popular LZW compression program	24
mtrt	A program that ray traces an image from Sun, we run in single thread	70

In our case, instruction trace is the sequence obtained in Kaffe JVM, since Java programs are executed on JVM. A commonly adopted method to get a bytecode trace of a Java class is to directly run Java program on JVM, and then modify JVM to collect Java bytecodes. In the experiments, by modifying kaffe [102], and running Java benchmark programs in interpreter mode, we get the runtime bytecode traces of all Java programs. These traces will be used as inputs in our simulator.

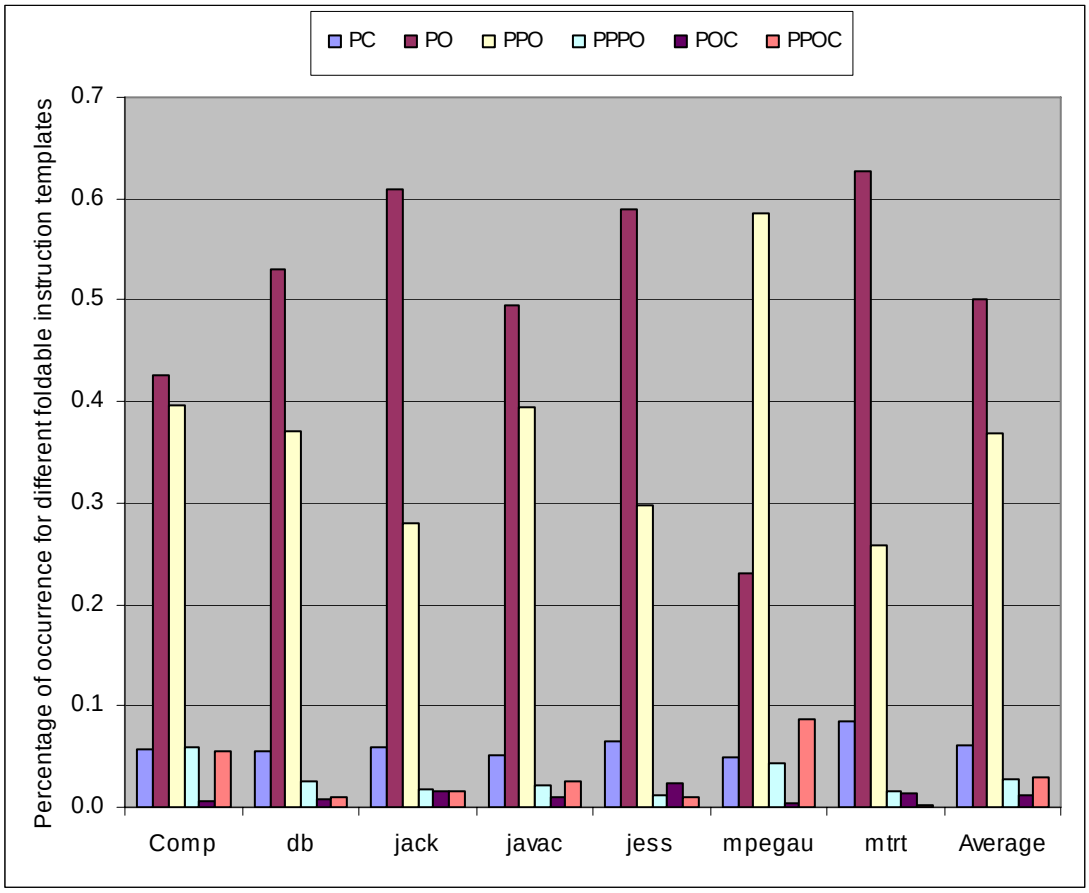
In the thesis, we run the benchmark programs from the SpecJVM98 suite [93] to evaluate the proposed stack folding scheme. We use the run-time traces collected from these benchmarks as our simulation input data. There are three input data set scales for the SpecJVM98 benchmarks: s1, s10, and s100. Here when using s1 data set, the benchmark programs will execute one time run. In this study, we run these benchmarks using s1 data set. Table 4.11 shows the benchmarks used.

4.8.2 Performance Results

The following simulation results are gathered from the trace-driven simulator when assuming the decoding rate at four bytecode instructions as designed in picoJava-II[88] and OTS as a 16-entry register file. In the simulation experiments, we did statistic analysis for the total 6 foldable bytecode instruction templates and calculate their distributions in total foldable instruction types as shown in Figure 4.4. The most occurrences are the *PO* and *PPO* templates, and they account up to 80% of the foldable instruction groups. The second most occurrences are the PC, PPPO templates, the least

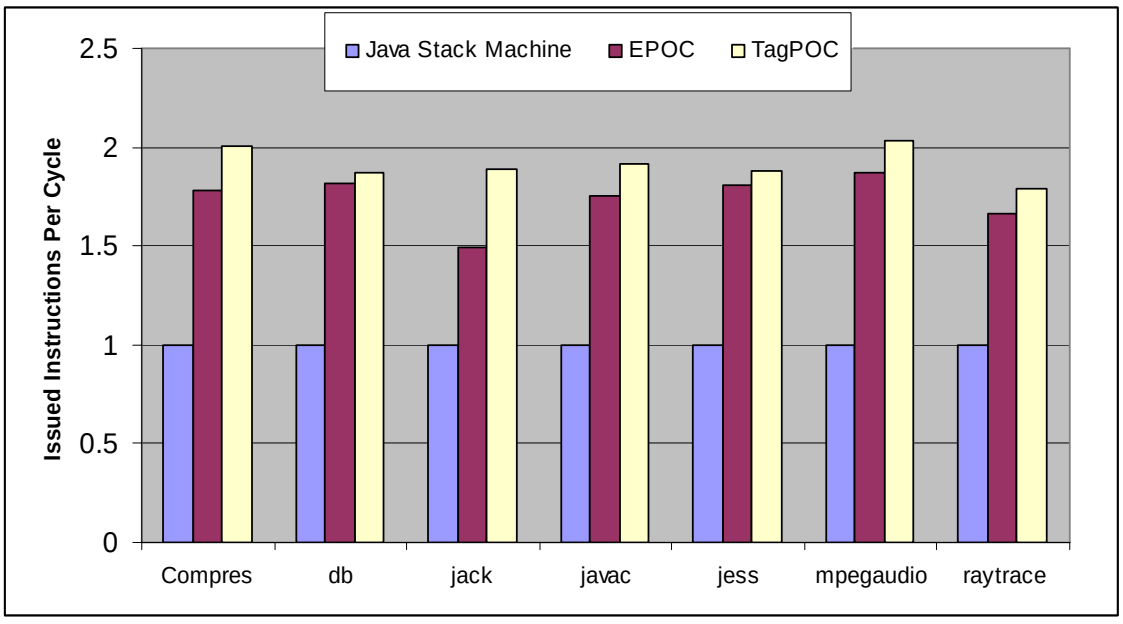
of them are the POC and PPOC templates. The distributions conform to the distribution of POC shown in [50]. It worth to notify that the distributions is dynamically generated, those foldable groups with intermediate generated results, such as OO patterns, are count to PO pattern. Because first O type instruction must finish the execution and only after its result is ready, the following foldable instruction group -- PO can be issued.

Figure 4.4. Percentage of different foldable templates occurred in benchmarks



The IIPC (Issued instruction per cycle) performance using different stack folding methods for a single-issued pipelined Java processor is shown in Figure 4.5. The IIPC performance with no stack folding, EPOC, and Tag-POC scheme are illustrated. Here we show the IIPC results of EPOC collected from [50] as comparison. The average number of IIPC in tag-based processor architecture can achieve as high as 1.745 which is slightly higher than EPOC-max which is reported as 1.74 [50]. This reveals that the tag-POC folding model can achieve the highest folding efficiency as compared to previous POC-based folding models.

Figure 4.5. IIPC performance for stack folding



Chapter 5

Exploiting Tag-based Abstract Machine Translator to Implement a Java ILP Processor

5.1 Overview

Chapter 4 introduced a stack instruction tagging mechanism, discussed how to implement a *tag-based abstract machine translator* (TAMT) for stack processors, proposed a tag-POC stack folding method. In this Chapter, we will use TAMT and tag-POC proposed in Chapter 4 to implement a Java ILP processor, and investigate some relevant issues.

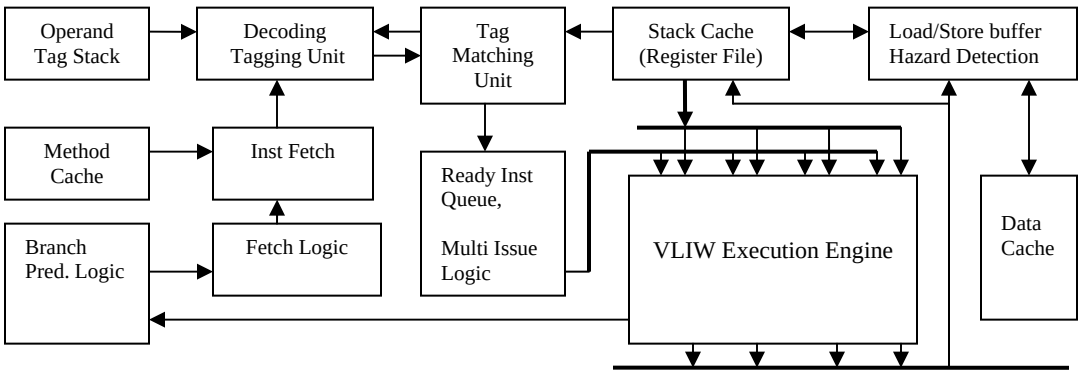
5.2 The Proposed Java ILP Processor

In the Java ILP processor, the real TAMT we used is organized by Tagging Unit (TU), Tag Matching Unit (TMU) and Operand Tag Stack (OTS). The OTS simulates the behavior of a real abstract stack machine and implements the execution of stack machine with tags. With instruction tagging, data dependences among the tagged

instructions are identified by tags. TU and TMU are responsible for controlling the instruction tagging, matching tags and updating the status of tags. When a bytecode instruction enters the decoding unit, TU assigns a tag to it. After the instruction completes its execution, the related tags are released to the free tag pool by TU, where the tags can be reused. The schematic block diagram is shown in Figure 5.1.

The Java ILP processor we created is a pipelined processor with a six-stage, including instruction-fetching, decoding, stack folding, issue, execute and commit stages. Because instruction folding is on the critical path of the pipeline [88], one individually decoding stage for instruction folding is created in the pipeline. With stack instruction tagging and folding, bytecode instructions are converted to the tagged register-based instruction formats. When the operands of a tagged instruction are ready, it is added to the ready instruction queue for scheduling.

Figure 5.1. The proposed Java ILP processor architecture



In order to achieve high performance with reduced hardware complexity, a VLIW execution engine is employed in our processor. Multiple tagged instructions are bundled out-of-order depending on data availability to form VLIW-like instruction words. The instruction bundles are put in the issue buffer and issued by the Scheduler in-order. Although the instruction bundles are issued in-order, at the time they are bundled, they may be not in program order. Hence, our processor worked in *multiple-issue semi in-order style*.

The Stack Cache as a register file is provided [28] in the processor to eliminate inefficiencies typically associated with stack-based instruction processing, and it stores the temporary results in the instruction execution as picoJava processor [28]. Here we assume RF has enough read-ports (RP), e.g. as a four-issue machine, RF has at least 8 RPs. Write-ports (WP) are also needed to receive execution results. The hardware mapping of tags to operands is on the critical path of the pipeline, so in our design RF will immediately signal the TMU when a register value becomes available. In an alternative design, the TMU only confirms the readiness of an operand, and the delivery of values occur directly between the RF and the execution engine, while freed tags are signaled directly from the RF to the TU. This may simplify the scheduling and issue logic. In the following, we will discuss the different functions for each pipeline stage and its design issues.

5.2.1 Instruction Fetch and Decode

The Java bytecode instructions are fetched from a method cache. The bytecode fetch logic controls to fetch the instruction from the same bytecode method according to the program counter values. After fetching all the instructions of a basic block, fetch logic selects the next basic block as predicted by the branch predictor. With Operand Tag Stack (OTS) support, the Decoding Unit (DU) including TU and TMU are together to handle both instruction tagging and folding. When one bytecode is decoded, an entry in TMU is allocated, and the tag number of the entry is assigned to the bytecode. The policy of tag allocation is first come first service (FCFS).

The entry in TMU is to hold the control-related information, which contains the left / right operand tag number, the valid bit, the status bit and the address of a physical register in Stack Cache. A mapping table is used to manage the mapping from tags to registers. The Register File (RF) is a global temporary storage, responsible for storing stack operands and local variables to speed up memory access. The organization of TMU is similar to that of a reorder buffer in superscalar processor [26], but TMU holds more functions than a reorder buffer does. Whenever a result is produced in RF, the corresponding tag number is simultaneously sent to TMU to update the instruction status and wake up the waiting consumer instructions.

The operands for the instructions may be loaded from stack cache (register file) or the data cache. LV variables and intermediate results are both allocated on the register file.

The stack intermediate results generated by ALU instructions can be directly written to the register file in parallel. The memory load operation, if it does not exist in load buffer, must load from the data cache.

5.2.2 Instruction Issue and Schedule

In picoJava-II, the operand stack and local variable access are distinguished as *stack disambiguation* [88] to enhance the parallel execution. S*tack disambiguation* can logically differentiate different access types. This optimization will increase the performance, and an additional bit along with each instruction is added in picoJava-II to mark if it is an access to the operand stack or the LV area [88]. The proposed processor also distinguishes the operand stack and LV access with the same method as in picoJava-II.

In our processor, LV variables are resident on the stack cache too. When a folded instruction finishes execution, its result is first written back to the physical register the "destined" tag points to. Within a basic block, only the last write, e.g. istore_x, updates the corresponding LV variable if there are multiple writes to the same LV. This is similar to the register renaming and resolves the data conflict in case of multiple-writes to the same LV variable. With instruction tagging, WAR (write after read) and WAW (write after write) data dependences are removed, because both operations will access different registers. Thus only the real data dependence – read after write (RAW) needs

to be considered. When a RAW conflict occurs, our issue logic may guarantee the later instruction containing the LV read operation cannot be issued until the previous instruction containing the write completes.

The memory access instructions, such as iastore, iaload etc, may issue out-of-order. Memory dependences between instructions are detected at run-time by the memory-hazard-detection logic, which consists of a load buffer, a store buffer and address comparator circuits [13]. Store addresses are buffered in an address queue (FIFO). The store buffer can be used to make sure that operations submitted to the memory hierarchy do not violate hazard conditions. A store buffer contains addresses of all pending store operations. Before an operation (either a load or store) is issued to the memory, the store buffer is checked to see if there is a pending store to the same address. New load addresses are checked with waiting store addresses. If there is a match, the load operation must wait for the store it matches.

5.2.3 Instruction Execution and Commit

The ready tagged instructions are first dynamically packed into VLIW-like wide words, put into an instruction issue buffer, then issued to the functional units on VLIW engine through a Quasi-crossbar [63]. The bundled instructions are issued in strict locked-step as in VLIW machines.

When a bytecode instruction completes, the result will be written back to register file or load/store-buffer if it is a memory access instruction, and the status of the related tags

are needed to be updated. For the memory access instructions, once the operation has been submitted to the memory hierarchy, the operation may hit or miss in the data cache. In the case of a data cache miss for a load instruction, it must be made to wait until the data is loaded from the external memory into the data cache.

When the status updating of a tag is completed, we say the tag is “committed” and can be returned to the free tag pool for later uses. The live period of a tag is from the time that the tag is assigned to an instruction to the time that the instruction is finished execution. When a tag is no longer used, it will be removed and released for later retrieval, unless it is retained. If retained, the tag remains live till it is explicitly freed.

5.2.4 Branch Prediction

Branch prediction is simply handled using tags. In the event of an incorrect prediction, the stack must be restored to the state just before the branch. This can be achieved by placing a branch marker on the stack, and whenever the stack is popped below the marker, the items popped off are saved on a buffer stack. If the prediction is confirmed, the marker is removed from the stack if no saving has occurred, and any items in the buffer stack above the marker may be erased. If the prediction is incorrect, the stack content above the marker is erased in case of no saving, and any items saved on the buffer stack are returned to the tag stack. If prediction within prediction occurs, the second branch is handled similarly. If the first prediction is incorrect, only items saved for the first prediction are brought back to the execution stack, since both the True and

False paths of the second prediction are incorrect. The incorrectly dispatched instructions need not be purged from the pipelines; instead, their tags are marked for purging so that the instructions' results will not be stored into the registers and have no impact on subsequent execution.

5.3 Relevant Issues

5.3.1 Tag Retention Scheme

In the proposed Java processor, a tag entry consists of many fields, such as tag number, available flags, destination tag, operation field, Left operand, Right operand, and value, etc. The information of these fields is used to manage tags. In general, Java objects are stored in data cache. To access them, the processor needs to load them from data cache to register file or stack cache. This will increase overheads on system bus. In case of cache miss, the load operation will cause processor delay. If we create a special structure, such as small register file to hold those high-frequency accessed variables, this will reduce the processor delay. In our implementation, we allocated some entries in TMU to hold them. Here we introduced a tag retention scheme to achieve this target. In the following we describe the tag retention scheme.

The solution to the repeatedly needed variable problem mentioned is to retain reused values by retaining their tags instead of freeing a tag after use. Suppose we want to compute: y=(((ax+b)x+c)x+d)x+e.

Here, the value of x is repeatedly needed. The compiler inserts, after the Load x instruction, a Retain instruction requesting that tag for the loaded value be retained as Retained Tag 1, and the tag allocated by the execution unit to represent the value, which is placed on the stack after the load instruction has been issued, and be copied to a free entry of the Retained Tags (RT) store. The compiler keeps track of which RT entry is being used for which variable. A retained tag on the stack is picked up by the consumer instruction in the usual way and the loaded value will be delivered to the consumer as usual. However, later uses of the value then require the compiler to replace the Load x instruction by the ReUseTag instruction with compiler providing the correct RT number for X, which would put the previously allocated tag to be read from the RT store and placed on the tag stack. The consumer instruction will pick up the tag and then retrieve the value from the physical register depending on whether we are using virtual or real registers. In our design, each register will have a Retain flag (the flag is also used in branch prediction since the value needs to be retained till prediction is confirmed). The Retain flag will be set when a Retain instruction is executed on a newly allocated tag which has just been placed on the top of the stack; and twill be cleared when a FreeTag instruction is executed while the tag is returned to the free tag pool and deleted from the RT store.

The cooperation of compiler and hardware to implement retainable tag will improve system’s performance, since the data locality allows for us to store data locally to

reduce data miss and putting them on register will reduce latency delay due to memory access. When we combine the tag retention with the management of LV variables, algorithms similar to compiler register allocation methods will be used to schedule retainable tags and reduce register spills. That is, in our system registers are allocated dynamically by the hardware tagging unit, but RT is managed by the compiler.

To implement tag retention scheme, we can modify Java compiler by adding another three new instructions. If applying this method, another hardware resource – Retainable Tag Stack (RTS) will be needed. These will increase the difficulty of implementation. Currently we did not want to involve the Java compiler design, so we implement the scheme with a simple method. We allocated some number of tag entries as retainable tag, which can be retained to hold retainable variables. However, it is needed to decide how many numbers of entries can be retained. In next chapter, we will calculate this number through experiments.

To make it simple, we add a field of retained status in tag entry. In the proposed processor architecture, the result of a producer or operator instruction is kept in the tag entry till it is consumed. If the result is consumed multiple times, the retained flag of the producer has to be set until the result is not needed anymore. This process is guaranteed by the syntax of stack machine. A consumer instruction will remove its operands from TMU and release them if no retained flag is set when it finishes execution. If a retained flag is set, its operands will be kept alive.

To implement tag retention scheme, we allocated some entries from TMU to hold retainable tags. Here we should set a maximum number of tags to hold retainable tags. Most of retainable variables are LV variables. When the index of LV_load instruction is less than the maximum number, it can be obtained from the retainable tag entries, otherwise, the load operation must load data from cache. This will incur one cycle of delay. And in later case, stack folding will detect it, and the folded RISC-like instruction will also be delayed one cycle until its dependent LV load is ready. The performance effects will be investigated in Chapter 6.

5.3.2 Memory Load-Delay in VLIW In-Order Scheduling

In superscalar processors, as we all know, though a program has instructions producing data in registers and consuming data from registers, during actual execution this producer-consumer relation is achieved through the common data bus (CDB) with instructions executing earlier or later than their order in the program depending on when the data become available. In particular, a cache miss would cause a delay in the completion of a load instruction, and consequent delays in instructions that consume the loaded value. The common data bus allows the buffering of such delayed instructions in the reservations stations so that the program executes correctly despite such unpredictable delays, but at the cost of higher hardware complexity as well as runtime overhead.

In VLIW processors, the work done by the common data bus at runtime is pushed to the compilation stage. That is, instructions are moved in the program to the correct positions such that upon execution instruction grouped, the data are immediately available in their source registers. Further, instructions are grouped into parallel bundles, each of which may consist of three instructions like in Itanium/EPIC processor, and all the instructions within the same bundle are guaranteed to be executable in parallel. The compiler tries to fill the slots of a bundle with instructions that are independent of each other, and with each received bundle, the execution unit simply pushes each instruction out to one of the execution pipelines. Since each pipeline might specialize in executing a particular type of instructions, there may be some restrictions on what instruction can be used to fill which slot of a bundle. This is reflected in the format of the bundle, sometimes known as a very long instruction word. (In some machines, a parallel bundle may consist of more than one VLIW instruction, which have flags set to indicate they belong to the same bundle i.e. in EPIC machine.) The detailed discussion on memory delay problem in the proposed Java ILP processor will be included in Chapter 8.

5.3.3 Speculation-Support

To exploit more potential ILP, to overcome the limitation of control dependence is needed, which can be done by speculating on the outcome of branches and executing program as if we guess correctly. Speculation mechanism extends over branch

prediction with dynamic scheduling. But at the same time, we need to handle the situation where the speculation is incorrect.

The hardware-based speculation is widely implemented in a number of processors, for example, PowerPC series, MIPS R10000 [43], Intel Pentium II/III/4 [24,51], Alpha 21264 [81], and AMD K5/K6/Athlon, etc. The implementation of speculation execution in these processors is based on Tomasulo [85] algorithm.

Commonly to speculative execution is to allow instructions to execute out of order but to force them to commit in order and to prevent other irrevocable action (such as register file updating or taking an exception) until an instruction commits. In speculative processors, the pipeline stage of completing execution is separated from instruction commit. And a reorder buffer (ROB) is often used to pass results among instructions that may be speculated. The ROB holds the result of an instruction during the period from the time the operation associated with the instruction completes to the time the instruction commits. In Tomasulo algorithm without speculation, once an instruction writes its result, any subsequently issued instruction will find the result in the register file. With speculation, the register file is not updated when the instruction commits. Thus, the ROB provides operands in the interval between completion of instruction execution and instruction commit. In commit stage, when a branch with incorrect prediction reaches in the ROB, it indicates that the speculation is wrong. The ROB must be flushed and the processor restart execution at the correct successor of the

branch. If the branch prediction is correct, the branch is finished. The commit phase is completed. The more detailed discussion about speculation techniques can be referred to [64].

5.3.4 Speculation Implementation

To further improve the performance, and exploit more potential ILP in the proposed Java ILP processor, we implemented speculative execution. We used a centralized ROB-based mechanism. In the processor, the TMU is used to store instructions and monitor the updates. In order to correctly recover and flush the speculated instructions, we add a structure, called *speculation tag buffer (STB)*, in TMU to record tag sequence of the speculated instructions. The structure may be a small amount of register file, and each entry is only one byte to hold a tag number.

When a branch instruction is predicted, its successor instructions are speculated fetched, and the corresponding tags are allocated from the tag pool in sequence order. These instructions are set a flag at each entry to indicate that they are speculated, and their intermediate results will be stored in corresponding register file. Until the branch instruction is confirmed, their speculated flags are reset, and they can be committed according to STB. If the branch prediction is wrong, those speculated tag entries in TMU will be flushed according to STB, the processor will restart from the correct instruction. Because the tags stored in STB followed in program order and CPU

commits speculated instructions in order, the commit process can guarantee the correctness of program execution. To recover from the wrong speculation, Operand Tag Stack (OTS) also needed to recover. When a branch instruction is encountered, the status of OTS will be stored in memory, which is the simplest way. If a recovery is needed, just copy the old OTS status from the memory, and restart tag execution based on the old OTS status.

In order to reduce the case of exception recovery, we limited the memory load operations cannot issue until its previous branch is confirmed. This design can reduce complex cache-miss induced memory exceptions.

Chapter 6
Performance Evaluation

We have done a simulation study on the proposed Java ILP processor architecture. The proposed Java ILP processor issues instructions in multi-issue semi-in-order style (as described in Chapter 5), so we called it **TMSI** processor. A trace-driven simulator was developed to model the TMSI processor's pipeline architecture. The simulator accepts bytecode traces extracted from the execution of the benchmark programs on the modified open source Java VM interpreter Kaffe [102]. The bytecodes are scheduled and executed on the simulator cycle-by-cycle. The algorithm of bytecode instruction tagging and management follows the processor model. In this chapter, we will evaluate the performance of TMSI processor.

6.1 Experimental Methodology

6.1.1 Trace-driven Simulation

Trace driven simulation is an important method of easily gathering performance statistics without becoming bogged down in the details of full simulation from an executable image [64]. Trace-driven simulation is efficient, because the simulator is

concerned only with the processor features that affect performance. To determine performance, the simulator simply models the functional units as elements which delay the operand values needed by some instructions and prevent simultaneous execution of some other instructions. Trace-driven simulation uses a predetermined instruction sequence and the instruction execution trace [64] to evaluate microprocessor performance. We developed a trace-driven simulator to analyze and evaluate the performance of our TMSI processor.

6.1.2 Java Bytecodes Trace Collection

A commonly adopted method to get a bytecode trace of a Java class is to directly run Java program on JVM, and then modify JVM to acquire and collect Java bytecode. Two popular JVM implementations can be used in this study: the SUN JDK and Kaffe VM 1.0.7 [102]. Both of the JVM implementations support the JIT and interpreted mode. Since the source code for the Kaffe VM compiler is available, we can instrument it to obtain the behavior of the class and then get the trace.

In this study, by modifying Kaffe [102], and running Java benchmark programs in interpreter mode, we get the runtime bytecode traces of the SpecJVM98 benchmark programs. These traces will be used as inputs to the trace-driven simulator.

6.1.3 Simulation Workloads

In the thesis, we run the benchmark programs from the SpecJVM98 suite [93] and Linpack [52] to evaluate TMSI processor's performance. We use the run-time traces collected from the benchmarks as our simulation input data. The SpecJVM98 benchmark suite includes Db, Javac, Mtrt, Mpegaudio, Compress, Jess and Jack. Their description and trace sizes are similar as shown in Table 4.10 and as for Linpack program, it is a computing intensive application used in many benchmark test suite for testing computer's performance. In this study, we run these benchmarks using s1 data set, which denotes that the benchmark programs only run once on JVM and the *Mtrt* benchmark program is a single-thread version.

These benchmarks do not include any graphic, networking or AWT, and therefore do not represent a whole spectrum of Java applications [88]. However, they do provide us with a starting point to evaluate the performance of the proposed Java processor architecture.

6.1.4 Performance Evaluation and Measurement

The performance evaluation of TMSI processor is done based on the experimented benchmark trace analysis. Here we introduce the speedup formula used to assess the performance gain.

The most indicative performance evaluation factor, the program execution time, is given by [41] :

$$ET = CPI * C * T \qquad (1)$$

Where ET is the execution time, CPI is the average clock cycles per benchmark instruction, C is the total dynamic instruction count, and T is the clock period. From the formula (1), we get the speedup as

$$\text{Speedup} = \text{CPU execution time}_{\text{before}} / \text{CPU execution time}_{\text{after}} \qquad (2)$$

In the study, we adopt the average CPI alone as performance evaluation measure. If we presume the usage of the same clock period (T) for the purpose of comparison, and dynamic instruction count is the same, the speedup formula can be reduced to:

$$\text{Speedup} = \text{CPI}_{\text{before}} / \text{CPI}_{\text{after}} \qquad (3)$$

In the thesis, all of places we will use the formula (3) to evaluate the performance gain.

6.2 Simulator Design and Implementation

In the experiments, a trace-driven simulator was developed to analyze the performance of TMSI processor. The simulator models a pipelined processor at cycle by cycle basis.

As is common as modern processors, the TMSI Java processor has a six-stage pipeline, including instruction-fetch, decode/tagging, tag/value matching, issue, execute and commit stages. After instruction tagging converting stack instructions into tag-based RISC-like instruction formats, a Tag Matching Unit (TMU) acquires the operands of the tagged instructions from the stack cache (register file) through the tag/value match window. If the operands are ready, the tagged instructions are added to the ready instruction queue and later bundled as VLIW instructions to be issued in-order to the VLIW execution engine. Although the instruction bundles are issued in-order, at the time they are bundled, they may be not in program order, depending on data availability.

Figure 6.1. Basic pipeline of TMSI Java processor

Inst-Fetch Fetch fixed Insts. to Inst-Buffer from I-Cache	**DECODE I** Decoding / tagging instruction	**DECODE II** Stack Folding & tag/value matching	**ISSUE** Packet bundles, Issue & Schedule	**EXECUTE** Execute for one or more cycles	**COMMIT** Write back results into Stack Cache or Update Memory

The developed trace-driven simulator executes the trace simulation according to the TMSI processor's pipeline cycle by cycle. The instruction execution pipeline is shown in Figure 6.1. In each cycle, Fetch unit fetches 4 instructions, and pre-decodes them. In ***Decoding stage***, instructions are tagged according to the tag management algorithm which follows the Java program execution paradigm, then after completed stack

instruction folding, a new tag-based RISC-like instruction format is generated. The newly generated instructions are scheduled and issued in ***Issue stage,*** but the limitations of instruction dependences must be obeyed. In ***Commit stage***, after completed execution the bundled instruction groups will update tags' status in TMU in order that the ready instructions could be scheduled in next pipeline cycle.

Table 6.1. Input parameters in the simulator

Fixed Parameters	
Processor pipeline	six-stage (F,DI,DII, Issue, Ex, WB)
Decoded instruction size	4
Instruction Issue-width	4
Size of TMU	64 entries
Data Cache Setting	Perfect cache
Instruction Cache	Ideal cache
Instruction cache size	enough to hold any class method
Variable Parameters	
Branch predictor	Static predictor (branch predictor penalty 3 cycles), for speculation cases, recovery overhead at 6 cycles.
A number of integer unit	2
A number of floating unit	2
A number of memory unit	2

In the performance simulation experiments, we assume the system has 2 load/store units, 2 integer units and 2 float-point units. We assume that TMU has 64 tag entries. The size of physical register file is larger than 64, because register file not only provides tag-mapping registers but also contains the LV storage area. And a static branch predictor is used, which is easily implemented by hardware and has a penalty of 3 cycles for mis-predicted branches. Additionally a perfect data cache is assumed as well as an ideal instruction cache was assumed to provide in the experiments. The detailed description of all assumptions and structure sizes are shown in Table 6.1

6.3 Performance Evaluation

We used SPECjvm98 [93] and Linpack [52] benchmarks. In the experiments, instruction schedule was limited within a basic block (except for speculation cases), only when all the instructions within a basic-block were issued can the instructions in the next basic-block be scheduled, but instruction prefetch is supported.

The proposed Java ILP processor issues instructions in multi-issue semi-in-order style (as described in Chapter 5), so we called it **TMSI processor**. To study the gain in ILP and performance speedup with TMSI processor, we ran two types of simulation: one in which every bytecode instruction assumes at a single cycle latency, and the other in which the different bytecodes take different latencies according to the picoJava specification. ILP gain is useful for determining ILP speedup from the viewpoint of multiple instruction issue, and the latter simulation is helpful to demonstrate the actual speedup compared with the existing architecture, and indicates the actual performance gain in TMSI processor.

6.3.1 Exploitable Instruction-Level-Parallelism (ILP)

To detect the proportion of parallel execution instructions in TMSI processor, we relax the resources constraints on the number of execution units and set the issue rate at four. When the execute stage is fed all the instructions within the instruction issue window, the processor could potentially execute at most four of them in parallel if there are no

dependencies and resource constraints. If there are stack dependences or LV dependences, the following instructions will be executed in the next cycle.

Table 6.2 shows the proportion of instructions execution in parallel in percentage when we run the benchmark programs on 4-issue TMSI processor. From the table, we obtained that a higher-percentage tag-based RISC-like instructions are executed in sequential, and the percentage of 2-issue, 3-issue and 4-issue instruction groups are less than 1-issue instruction groups. This is determined by the characteristics of the Java benchmark programs. And from the table, we can see that for the *mpegaudio* benchmark program, the 4-issued instruction groups are in higher percentage, which is different from the other programs. The reason is that it is computing-intensive program and the size of the average basic-block is larger than others, thus more instructions can be executed in parallel. Further, in *mpegaudio* benchmark, more than 30% issued instruction groups belong to 4-issue group, which caused a better performance gain.

Table 6.2. Percentage of instructions executed in parallel in our scheme

Benchmarks	**Tagged instructions executed in parallel (percentage)**			
	1	2	3	4
Compress	67.37	15.43	10.78	6.42
Db	79.97	14.98	3.78	1.27
Jack	79.54	14.22	3.89	2.35
Javac	72.85	21.87	4.24	1.04
Jess	81.51	13.47	3.26	1.76
Mpegaudio	43.26	16.53	6.78	33.43
Mtrt	87.92	9.67	1.55	0.86
Linpack	69.18	16.10	0.38	14.34

Table 6.3. Percentage of instructions executed in parallel using stack disambiguation

benchmark	Instructions executed in Parallel				
	1	2	3	4	5
db	78.76	21.06	0.17	0.00	0.00
javac	81.25	18.17	0.57	0.00	0.00
jess	81.06	18.20	0.74	0.00	0.00
mpeg	87.32	12.47	0.21	0.00	0.00
mtrt	86.44	11.56	2.00	0.00	0.00

Let us compare the result in Table 6.2 with that reported in the previous research work of in-order multi-issue of the folded Java instruction execution [88] (shown in Table 6.3). In this report, by using stack disambiguation technique, only a small number of three-instruction-groups are issued in parallel and no four-instruction-groups are issued in parallel. Compared with it, you can see that the tag-based method can explore more ILP in Java programs. However, the results of our experiments show that the percentage of issued three-instruction-group is from 0.3% to 10%, and the percentage of issued four-instruction-group is from 0.8% to 14%, except *mpegaudio*. The percentage of *mpegaudio* is higher up to more than 33%. The reason is that the basic block of *mpegaudio* is bigger, and within a basic block there are more ALU instructions which can be run in parallel. These results demonstrate that ILP is enhanced in our TMSI processor.

In order to investigate the perfect ILP within a basic block in the benchmark programs, we do the following assumptions: the first is the decoding rate is set at 4 similar in Table 6.2, and the second is we assume that no resources limitation for the instruction

execution, that does mean when those decoded instructions are ready in issue queue, all of them can be issued; the third is we set the maximum instruction issue rate at 8. Based on these assumptions, we re-executed the benchmark programs. Table 6.4 shows the result for this case.

In Table 6.4, we can see that most of instructions within a basic block can be issued within 4 issue-groups. Even though we relaxed the instruction issue limitation for resources, only small amount of basic-block instruction groups can issue instructions more than 4. For most of benchmark programs except c*ompress and mpegaudio*, only less than one percentage of instruction groups can issue instructions more than 4. For *compress* benchmark, the number of percentage is 3.16% and for *mpegaudio* benchmark, the number of percentage is 26.41%. Compared with Table 6.2, we can see that although the more resources can be added in the Java processor, the very less ILP improvement can be obtained. Thus, if we consider the hardware complexity and pipeline execution efficiency, we prefer a 4-issue Java ILP processor.

Table 6.4. Percentage of instructions executed in parallel with unlimited resources

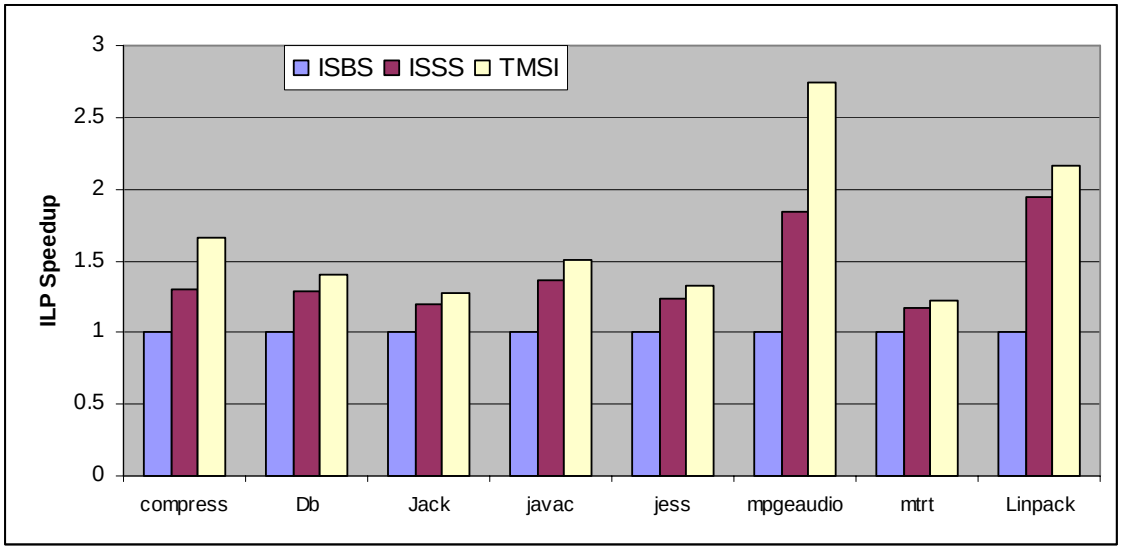
Benchmark	Tagged instructions executed in parallel (percentage %)							
	1	2	3	4	5	6	7	8
Compress	62.6	13.7	17.1	3.4	1.8	0.3	0.04	1.02
db	79.1	15.0	4.6	1.04	0.12	0.09	0.03	0.001
jack	75.8	17.5	4.18	1.96	0.12	0.27	0	0.11
javac	71.2	21.3	6.35	0.78	0.02	0.27	0.03	0.03
jess	80.9	13.6	3.73	1.1	0.2	0.27	0.16	0
mpegaudio	45.1	18.4	5.9	4.16	5.75	9.62	2.4	8.64
mtrt	85.6	11.87	1.62	0.77	0.02	0.01	0.03	0.00
linpack	45.98	36.06	0.79	15.81	0.45	0.9	0	0

6.3.2 ILP Speedup Gain

To compute the ILP gain, we assume all instructions with unit latency. Figure 6.2 presents the ILP speedup results for three different configurations: in-order single-issue base stack (ISBS) processor, in-order single-issue with stack folding stack (ISSS) processor and our multi-issue in-order TMSI processor. The stack folding used in the experiments also supports nested folding. With the tag-based stack folding scheme, the ILP gain for ISSS processor can be seen from 20% to 90%. This result demonstrates that the tag-based stack folding scheme is effective, particularly for computing-extensive cases, such as *Linpack* and *mpegaudio*. The ILP gain with TMSI multi-issue over ISSS stack processor is also observed to range from 3% to 27% for all applications except *mpegaudio*, for which the gain is 49%. The result also demonstrates that TMSI Java processor can improve the performance than ISSS stack processor does. The ILP gain with TMSI processor over ISBS stack processor can be seen from 21% to 115%,

except for *mpegaudio* case in which the gain is 173%. This shows that the ILP speedup can be obtained through both stack folding and multi-issue in Java processors.

Figure 6.2. ILP speedup gain: TMSI vs. base Java stack machine

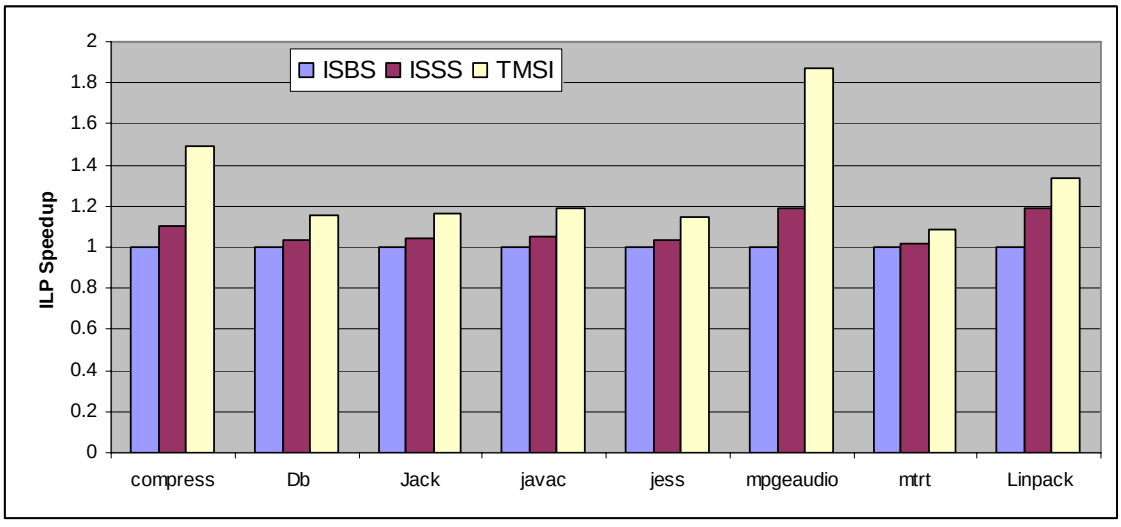


6.3.3 Overall Performance Enhancement

Figure 6.3 demonstrates the actual speedup obtained using the varied latency according to the picoJava-II specification. With the configuration of in-order single-issue with stack folding stack (ISSS) processor, an improvement of 2% to 19% is observed. With multiple-issue TMSI architecture, the speedup ranges from 9% to 34% for all applications, except *mpegaudio* and *Compress* benchmarks. The actual speedup of *Compress* program with TMSI is 49% while the actual speedup gain of *mpegaudio* is 86%. The reason for the much higher performance speedup observed in *mpegaudio* is

that more bytecode instructions are executed in parallel than in other benchmark programs. Compared with SMTI [79] processor, the results obtained are as good as or even better than those in SMTI, except *mpegaudio* benchmark. This result demonstrates that our tag-based mechanism can exploit more ILP. For *mpegaudio* benchmark program, software-implemented multi-trace SMTI [79] processor may schedule instructions within a bigger instruction window than our scheme when bigger basic blocks exist. In contrast, the instruction schedule window in TMSI processor is constrained by the size of TMU. However, our architecture does not need complex fetch logic to support.

Figure 6.3. Overall speedup gain: TMSI vs. base Java stack machine



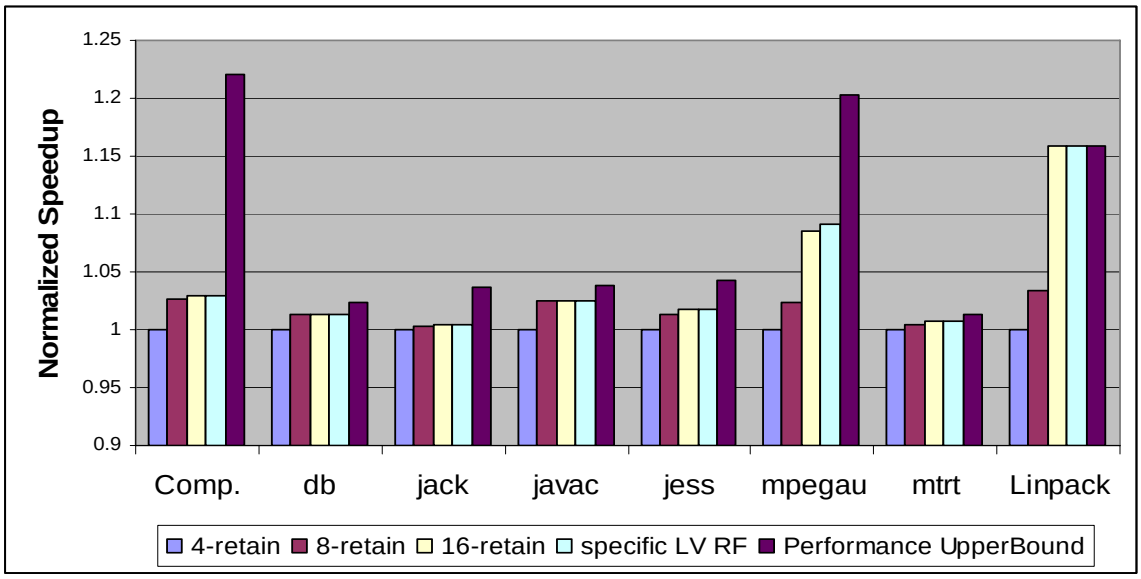
6.3.4 Performance Effects with Tag Retention

In this section we will investigate the performance effects when different number of retainable tag is allocated from TMU. Here we assume the processor did not allocate on-chip register file for LV (local variable) variables' use (here is different from the previous configuration.), and when a LV variable will be used sooner, it will be set with a retention flag. This procedure is dynamically implemented. After the tag is accessed or used by later consumer instruction, it will be released. In order to demonstrate the performance effects, we had the following assumptions: some tag entries from TMU will be allocated and the maximum retainable number (MaxRet) of tags is assumed at 4, 8, and 16 accordingly. When a LV load access instruction is encountered, if its LV number is less than MaxRet, the value is accessed from the on-chip register file since the tag entry is retained from TMU; otherwise, it will be loaded from the data cache. When load from the data cache, an extra instruction load cycle is needed, and the later stack folding operation must delay one cycle until the load is ready for access. For LV store instruction, since a memory store buffer is provided we don't consider the access delay and if a LV store is immediately needed for a LV load, the direct data forwarding is provided to reduce the performance lost.

Figure 6.4 illustrated the performance effects when allocating different number of tag entry (SizeRet) at 4, 8, 16 as retainable tags from TMU. In the mean time, for the comparison purpose, we also gave the normalized speedup performance when allocating a special-purpose LV register file and as well the performance upper-bound

when no any instruction issue limit exists. In the experiments, bytecode instruction latencies follow those defined in picoJava-II specification. The experiments demonstrated that when SizeRet is set at 4, 8, 16, the effects of performance gains are from 0.5% to 3%, except for the benchmark program *Mpegaudio* and *Linpack*. The effect for the *Mpegaudio* benchmark is at 8%, and for *Linpack* program is at 15%. This is because these two benchmark programs contain a large number of LV access instructions for use of intermediate variable. For *Linpack* program, the maximum number of LV needed is 48, and for *Mpegaudio* program, the number is 38. (We obtained them from an analysis of the bytecode traces obtained from executing SpecJVM98 benchmark suite.) For the other benchmark programs, the maximum number of LV is around 20, thus the performance effects is smaller in these benchmark programs and when the SizeRet is set at 16, the performance gains for them nearly reach the same value as using a specific-purpose LV register file.

Figure 6.4. Normalized speedup with different amount of retainable tags



However, another advantage of adding a specific-purpose LV register file is mainly for speedup the execution of procedure calls. In Java SpecJVM98 benchmark suite, procedure calls cost much more execution cycles, thus in order to increase performance and reduce the movements of reference data, a specific-purpose LV is needed [87].

6.3.5 Performance Enhancement with Speculation

In order to further investigate the possibility of performance gain, we implemented the speculation-support for TMSI processor simulator. In this section, we will show the performance results when scheduling tag-based RISC-like instructions speculatively. In the experiments, TMSI processor may schedule and execute instructions across more than one basic block. For the purpose of comparison of the performance gain when

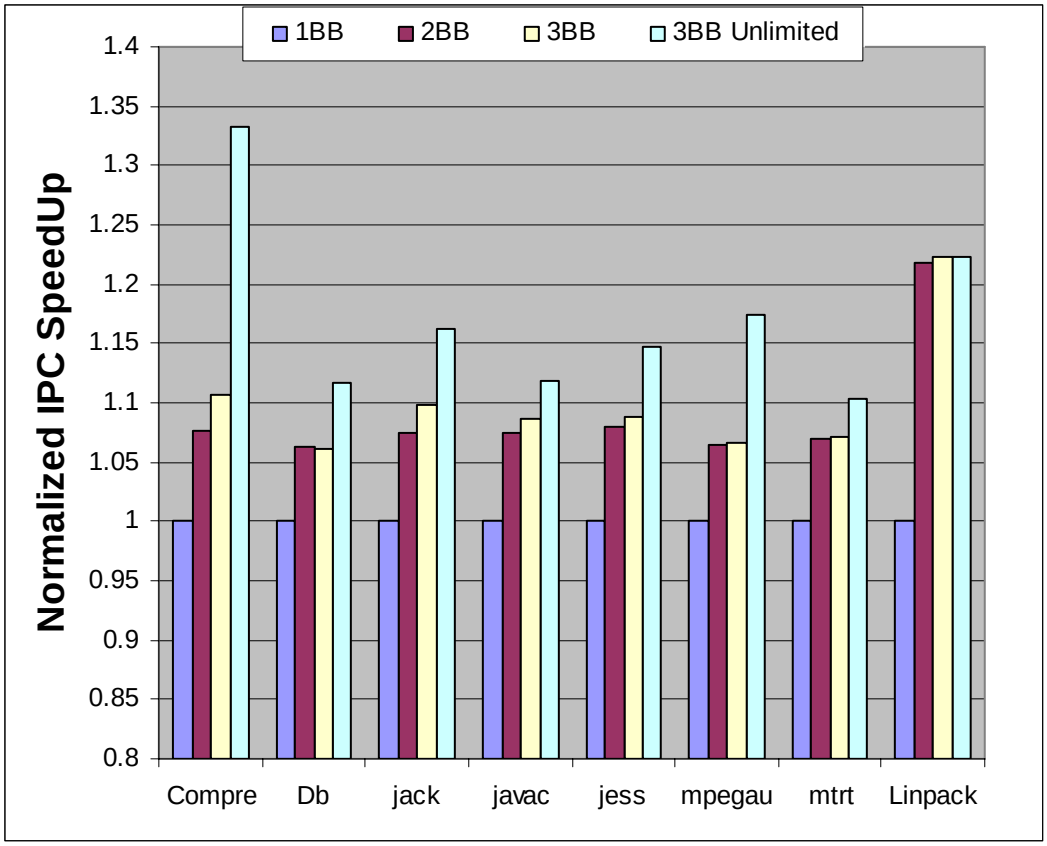
scheduling instruction across different instruction window sizes, TMSI processor simulator may dynamically schedule instructions across one basic-block (1BB), two basic blocks (2BB), and three basic blocks (3BB) accordingly.

In the simulation experiments, the simulation workloads are same as the previous experiments. Even though the execution trace for the benchmark programs are not in speculative dynamic scheduling mode, it may be accepted that the simulator re-schedules instruction traces in speculative mode to obtain the performance gains. Because we can not modify Kaffe JVM to execute Java programs in speculative mode. We assumed the TMSI processor only has one branch prediction unit and the recovery overhead is assumed at 6-cycle latency when the branch predictor predicts a wrong path. The detailed structural configuration can be seen in Table 6.1.

Figure 6.5 demonstrated the performance enhancement for speculative instruction scheduling when the instruction latencies are assumed with picoJava-II latencies, and in the experiment we use a BTFN (Backward Taken Forward Not-Taken) static branch predictor, which is simple for a limited hardware complexity. The configuration of the used speculative TMSI processor is listed in Table 6.1. The results show that the performance gains when scheduling across two basic blocks can be seen from 6% to 8%, except for *Linpack* programs at 21%. This is because for SpecJVM98 benchmark programs, there are a lot of long latency instructions which are needed to be implemented with micro-codes. These instructions will affect the IPC (issued

instruction per cycle) performance gain largely. In additional, since *Linpack* benchmark aims to solve a matrix multiplication problem and it is computing extensively, this characteristic makes *Linpack* achieve better IPC performance speedup when using speculative scheduling in speculative TMSI Java processor.

Figure 6.5. Normalized IPC speedup with speculation scheduling



However, we also found that when the instruction scheduling is limited within three basic blocks (3BB), the IPC performance gain achieves to the limited value. This

limitation is due to the limited execution resources. In order to prove this, we have also done the simulation and obtained the IPC performance results (listed in 3BB unlimited column) when assuming that enough execution resources are provided for the 4-issue TMSI Java processor and the instruction scheduling is limited within three basic blocks. We found that if enough functional resources are provided, the IPC performance gain for 3BB unlimited case is much bigger than that of 3BB with limited resources for some benchmarks. Thus, we suggest that if the TMSI Java processor will support speculative instruction scheduling with limited hardware complexity, we prefer to constrain the instruction scheduling only across two basic blocks.

In the experiments, we also calculated the efficiency of the static branch predictor. We used BTFN static predictor in our simulation experiments. Here we employ static predictor in the experiments, the reason is that it is easy to implement in hardware with little complexity. Table 6.5 gives the basic static branch predictor's effectiveness for conditional branch and total branch predictor's effectiveness. The following two equations show how to calculate these two values.

Equation 1:

The efficiency of basic conditional branch predictor

= the number of branch hit / total number of conditional branch

Equation 2:

The efficiency of total branch predictor
= (the total number of unconditional branch + the total number of conditional branch hit) / (the total number of unconditional branch + the total number of conditional branch)

Table 6.5. Branch predictor effectiveness

Benchmarks	Uncondi-tional	Conditional			Total Efficiency (%)
		Branch Hit	Total Conditional Branch	Conditional branch Efficiency (%)	
Compress	103355	744397	1237116	60.17	63.24
Db	12668	188503	207973	90.64	91.17
Jack	531996	3733440	4223854	88.38	89.69
Javac	93961	763324	769087	99.25	99.33
Jess	83798	1111330	1226675	90.59	91.19
Mpegaudio	472328	3229690	3928163	82.22	84.13
Mtrt	773317	3582737	4967549	72.12	75.88
Linpack	409181	482636	494115	97.68	98.73

From Table 6.5, we can see that the branch prediction efficiency for most programs can achieve up to 90%, this demonstrated that in the used benchmark programs, we use BTFN static predictor is efficient for speculative scheduling cases. A high efficient

branch predictor may contribute to a very good performance for TMSI processor, and reduce the total recovery overhead due to speculative executing a wrong branch path.

6.4 Summary of the Performance Evaluation

A new approach of exploiting the concept of Abstract Machine and dataflow to extract Java-ILP has been illustrated in this Chapter. With instruction tagging mechanism, the independent bytecode instruction groups with stack dependences are identified. Because there is no stack dependence among the different bytecode instruction groups, they can be executed in parallel based on different register segments for multiple operands access and thus more Java-ILP is exploited. Based on the instruction tagging scheme, we proposed the TMSI Java ILP processor. In the processor, a TAMT are employed to translate bytecode instructions into tag-based RISC-like instructions, then execute them on a VLIW engine.

The simulation experiments demonstrate that the proposed TMSI processor architecture is able to significantly increase the average ILP over a single-issue Java processor. We calculated the geometric mean of the ILP and that of actual gain in speedup over all the applications, the results showed that the ILP gain is 59% and the actual speedup gain is 28% when the instruction latency is set as the defined in PicoJava-II specification. Java instructions defined in JVM include some complex instructions, such as *invoke* instructions, array access instructions, and method variable access instructions, etc.

These instructions often need a micro-code sequence to support, and conduct long instruction execution latency, further more they will cause the pipeline stall. These make the better ILP gain in the system not be fully translated into the real speedup.

Besides that, we also investigated the ILP performance when using tag-retention scheme and performance results with speculation technique which allows instruction schedule across more than one basic block. These results are useful for the further research on the proposed TMSI Java processor.

Chapter 7
Tolerating Memory Load Delay

7.1 Performance Problem in In-Order Execution Model

Instruction execution in traditional VLIW processors is exactly in-order. In-order instruction execution has severe IPC performance limits due to its inability to allow execution to continue past an instruction with an outstanding register use, where the register is being produced by a long latency instruction currently executing [55]. In this situation, the whole front-end of the processor stalls and it cannot issue any more instructions until the oldest instruction in the issue window obtains both of its operands. This kind of long pipeline stalls will degrade the performance of the in-order VLIW processors. The same situation often occurs on a memory load instruction, for example, when the memory load encounters an unpredictable cache miss. A Pending Functional Unit (PFU) [55] scheme is proposed by Lori Carter, which is devised to make EPIC / VLIW execution out-of-order in a small range to mitigate the performance effects due to unpredictable memory load delay.

Our TMSI Java ILP processor exploits a VLIW engine to execute tag-based instructions in-order, so the overall performance of the Java processor is also affected by such unpredictable memory load instructions due to data cache miss. In this chapter we will propose a new technique --- tag-based PFU (tag-PFU) scheme -- to mitigate the performance degrading. Our aim is to hide or reduce the effects of unpredictable long latency load instructions, without adding a large amount of additional hardware complexity.

In the following we look at the implementation techniques in superscalar processors from the comparative perspective, describe our tag-based PFU scheme and evaluate the performance.

7.2 Out-of-Order Execution Model

The high-performance processors generally adopt Tomasulo [85] scheme to achieve out-of-order execution. However, the major drawback of the approach is hardware complexity. In particular, the use of reservation stations requires complex control logic [32]. Lastly, the performance can be limited by common data bus (CDB). The critical points in Tomasulo scheme are dynamic scheduling, register renaming and dynamic memory disambiguation. The register renaming plays an important role in avoid data conflicts. Register renaming eliminates write-after-write (WAW) and write-after-read (WAR) dependences, but all the read-after-write (RAW) dependencies are preserved,

which are necessary for correct computation [41]. Renaming extracts the maximum parallelism from an application since only necessary dependencies are retained.

Modern out-of-order processors are aggressive and optimistic. They always try to execute multiple instructions per cycle speculatively. Similar aggressiveness is also observed in execution of memory load operations and instructions that depend on these memory loads. In most processors, instruction scheduling is based on the assumption that the load hits in the cache. This assumption usually increases the performance as most of the loads actually hit in the cache. However, if an instruction is dependent on a memory load that misses in the cache, those dependent instructions will need to be re-executed. This re-execution is referred to as *replay*. Two methods can be employed to handle this situation: flush replay (used in Alpha 21264 [81]) and selective replay (used by Pentium 4 [24]). In flush replay, all instructions in the issue window are flushed and re-executed whatever they are related to the load operation or not. In selective replay, the processor only re-executes the instructions that depend on the missed load.

An out-of-order pipeline aims to execute an instruction as soon as it is ready, not according to some predetermined order that may be not efficient based on the run-time conditions. The out-of-order processors often make use of the available run-time information and schedule instructions dynamically to overcome the unpredictable long latency of load memory delay due to cache misses. If a memory load instruction misses

in the cache, future instructions dependent on the loaded value must wait, but other instructions that are ready may proceed [19].

Most out-of-order processors schedule instructions can across multiple basic blocks, and implement branch prediction and speculation execution. Even though instructions are executed out-of-order, the results are committed in-order. This keeps a sequence execution model and guarantees the correction of the program execution. To keep track of the original order that instructions entered the pipeline, a FIFO structure called *reorder buffer* (ROB) is used. The speculation instructions and state are kept in ROB, when a branch is mispredicted, the recovery is easy to implement by clearing the ROB for all entries that appear after the mispredicted branch instruction, allowing those that are before the branch instruction in the ROB to continue. Furthermore, with this architecture, a precise exception may be implemented.

To summarize, the order of instruction execution for an out-of-order execution machine is determined by the hardware dynamically, and the run-time information need to be taken into account, and accommodations could be difficult to make due to unpredictable cache misses.

7.3 VLIW/EPIC In-Order Execution Model

VLIW / EPIC machines achieve instruction level parallelism (ILP) due to their ability of issuing multiple instructions operation per cycle and with relatively simple control logic. They issue instruction bundles in-order. In-order execution processors may suffer an expensive stall when servicing data cache miss. This problem is exacerbated because the data cache miss shows hard-to-predict. To effectively hide cache miss latency for in-order execution processors, micro-architecture enhancements as well as software optimizations can be applied. The compiler can insert prefetch hints into the programs to reduce data cache miss, or data caches are constructed as non-blocking caches to avoid unnecessary processor stalls.

In this chapter we will concentrate on the efforts of micro-architecture enhancement in tolerating memory load cache misses in a VLIW in-order processor. Rau [13] suggested the idea of small-scale reordering on VLIW processors to support object code compatibility across a family of processors. The Itanium processor (IA64) [33], an implementation of EPIC architecture, is an in-order processor which instruction scheduling is predefined by the based compiler. When Itanium pipeline encounters a memory load cache miss, the whole pipeline must stall to wait until the missed load instruction is finished. This will make its performance suffer significantly when small amount of cache is provided. To solve this performance issue, that is, to reduce its performance degrading in IA64, some approaches have been proposed. Perry et al [75]

proposed two approaches: one using an out-of-order (OOO) execution core, and the other assuming multithreading support and exploiting cache pre-fetching via speculative pre-computing. But the relative hardware issues involved in implementing the two approaches are needed to be considered. Another scheme is proposed by Lori Carter [55] called Pending Functional Unit (PFU) scheme to mitigate the performance effects with out-of-order instruction groups with a small range.

In thesis, TMSI Java processor employed a multiple-issue VLIW execution engine to execute instruction bundles in-order, but it encounters the same problem of memory load delay caused by unpredictable cache miss. We implemented the same function as PFU on the TMSI Java processor which is able to schedule other ready instructions first that are not dependent on the memory load instruction in order to mitigate some effects due to the memory load delay, we call it as tag-PFU scheme. Different from the PFU scheme, the proposed scheme did not increase any hardware complexity on the current TMSI Java processor, only through modifying the tag management algorithm. For the purpose of comparison, we first describe the implementation issues of PFU in IA-64 proposed in [55], and then illustrate our tag-PFU scheme implemented in the TMSI Java ILP processor.

7.3.1 PFU Scheme

IA-64 CPU is an in-order processor, which fetches, executes and forwards results of instructions to its functional unit in-order. The architecture of IA64 heavily relies on

the compiler to expose ILP to avoid stalls created by in-order processing. In IA-64, each cycle can have up to 6 instructions (2 bundles) scheduled to proceed on the pipeline and begin executing together [29]. If there is an outstanding dependency within a number of the scheduled groups, all the instructions in the group will stall and wait until all the instructions in the scheduled group are ready to start executing at the same time. The functional units in IA-64 also provide the bypassing logic which allows the values being produced to be directly consumed by another functional unit in the next cycle.

The basic idea of PFU [55] is to expose a small window of instructions, which have been allocated functional units, to be executed out-of-order. Instructions within IA-64 with PFU are issued exactly the same as in the traditional in-order VLIW architecture -- if there are WAW dependencies among instruction bundles, they cannot be issued. The hardware implementation of PFU is similar to *reservation stations* [85], but is simpler in that no scheduling needs to be performed when the operands are ready and the instructions already owns the functional unit it will use to execute.

When forming a schedule for the IA-64, the dispersal stage does not need to take into consideration the resource constraints. Thus, when the whole scheduled group goes to the functional units, the instructions either all stall together, or start executing together. However, IA-64 with PFU must take into consideration resource constraints for functional unit, and perform instruction scheduling in-order.

In next section, we will discuss how to implement the PFU function in TMSI Java processor, and compare design issues in micro-architecture. From the perspective of micro-architecture, our scheme may not only be implemented with less hardware complexity but overcome several drawbacks in PFU scheme. In this sense, the tag-based architecture can be extended and applied to other processor architectures.

7.4 Tag-PFU Scheme

7.4.1 Architectural Mechanism

The TMSI Java processor we proposed uses a VLIW engine to execute tag-based instructions in-order. It encounters the similar memory load delay problem as tolerating the effects of un-predictable memory load cache miss. The tag-PFU scheme in the thesis implemented similar function as PFU [55].

In TMSI Java processor, the scheduled groups are formed at instruction issue stage dynamically. The instruction groups are issued in-order but the sequence of individual instruction can be out-of-order. When the operands of an instruction are ready, the instruction is listed in the ready queue to be scheduled in the next cycle. If there are no any dependencies (RAW, WAW, WAR) within some ready instructions, they can be formed as a scheduled group to construct an instruction bundle. If there is RAW dependence, the instruction containing *read* operation must wait until the previous

instruction containing *write* operation finishes. The TMSI processor also has the capability of bypassing instructions, but the value produced by previous instruction is directly sent to register file instead of functional units. In TMSI processor, therefore, Tag-Matching-Unit (TMU) will play the role of buffering and scheduling instructions.

Because the instruction bundles in TMSI Java processor are dynamically generated, those instructions depending on the memory load could be buffered in TMU, delayed to issue until the memory load is finished. During that time, the other instructions that do not have data dependencies on the load can continue to be scheduled. This scheduling scheme can achieve the same function as in the PFU scheme, it can execute instructions out-of-order within a small range to mitigate the memory load latency caused by unpredictable data cache miss. Here TMU buffers those instructions which operands are dependent on the memory load and not ready yet at issue time.

In out-of-order processors, a *scoreboard* technique [41] is widely used to detect and maintain dependence information. A scoreboard may manage the issuing and completion of instructions or stalling of the pipeline based on operands and functional units being ready and dependences being met. Similarly, scoreboard can be cable to determine that the conditions are right for an instruction to execute in in-order processors. TMU in TMSI processor also works with a scoreboard function to determine whether an instruction is ready for issue.

7.4.2 Architectural Comparison

The PFU implemented in IA-64 exploits the structure of reservation stations (RSs) to buffer instructions depending on the result of the memory load instruction. The scheme has a disadvantage that needs to be addressed. If the cache miss occurs at the second level cache, the latency will be longer. In that case, a lot of dependent instructions should be suspending on the functional units, where they must be queued in the attached RS of the functional units. Under certain conditions, there are not enough RS entries provided (often, several entries are provided for each reservation station due to hardware complexity), then the overflow of reservation station happens. In this case, the pipeline will have to either stall or process the overflow. To solve this problem, extra hardware circuit is needed, which will add hardware complexity. In extreme cases, for example, if the memory cache miss occurs very often, the pipeline will have to stall frequently to wait for the load to be finished.

In contrast, our tag-based architectural mechanism may suspend those instructions dependent on the load results in TMU. That is because TMU plays a role of central ROB and can hold much more entries than RSs. What's more, given that in-order instruction schedule has seldom considered the availability of functional resources the tag-PFU scheme needs to consider this issue without increasing hardware complexity.

7.5 Effectiveness of Tag-PFU Scheme

7.5.1 Experimental Methodology

As seen in chapter 6, we have developed a trace-driven simulator which models the TMSI processor architecture. To simulate the memory cache miss, we integrated a cache simulator Dinero [31] into our simulator to evaluate the effects of data caches on the system performance. Dinero is widely used to analyze the cache performance.

In the program execution trace, we recorded all the memory access addresses to conduct cache simulation. We assume the system cache has two levels. The first level is directly mapped, and the second level cache is unified set-associative cache. We chose different sizes of data cache at first level from 1KB, 2KB, 4KB to 8 KB, 16KB, and 32KB to test their performance on the TMSI, whereas the second unified cache is assumed at 1MB. Here the cache replacement policy is supposed to use LRU algorithm. In the experiments, only memory load cache miss is considered. As for memory stores, the stored data is buffered by the Load/Store unit so we did not considered them. The instruction latency used in the experiments follows the picoJava-II specification and the data cache miss latency was assumed at 10 cycles at the first-level cache, and 50 cycles at the second-level cache.

In the performance simulation experiments, we used SPECjvm98 [93] benchmarks and executed them with the s1 data set, and instruction schedule is limited within a basic

block, but instruction prefetching is supported. Because the benchmark program – *Jack* can not smoothly complete execution with memory address trace flag on Kaffe environment, we only run the other six benchmark programs. In the experiments, the branch predictor used is a BTFN static predictor with 3-cycle penalty when a wrong branch prediction result is given. The detailed input parameters used in the simulator are shown in Table 7.1.

Table 7.1. The detailed input parameters in the simulation experiments

Fixed Parameters	
Processor pipeline	Six-stages (F,DI,DII, Issue, Ex, WB)
Decoded instruction size	4
Instruction Issue-width	4
Data cache size	First level: 1k,2k,4K,8K,16K,32K, LRU Second Level: unified 1Mbyte
Cache Miss Latency	First level: 10 cycles Second level: 50 cycles
cache mapping method	First level 4-way set-associative
Variable Parameters	
Branch predictor	BTFN static predictor (3-cycle penalty)
A number of integer unit	2
A number of floating unit	2
A number of memory unit	2
Instruction cache size	Perfect cache

7.5.2 Performance Results

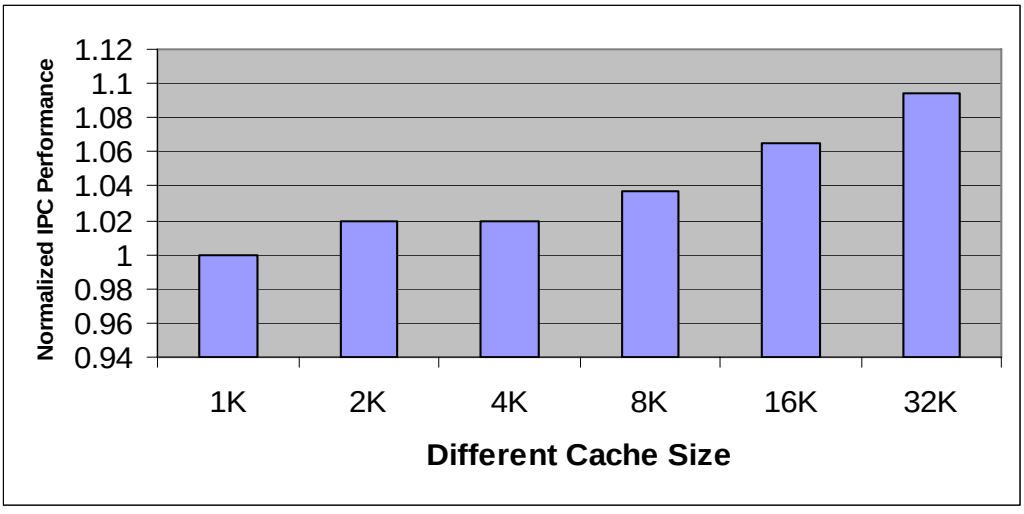
In the following, we will illustrate the performance and cache simulation results for each benchmark program.

7.5.2.1 IPC Performance with Different Cache Size

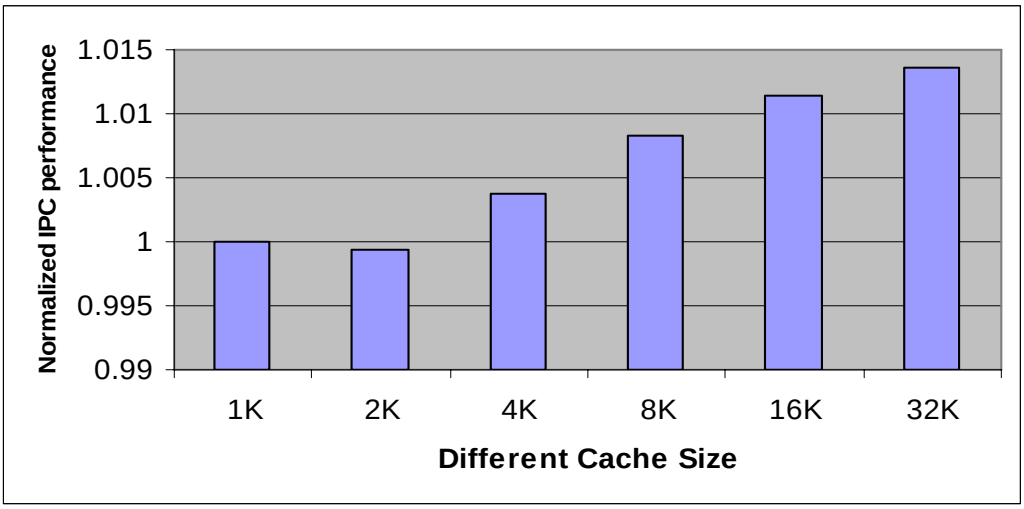
In Figure 7.1 (a) – (f), the performance simulation results for the benchmark programs are shown as cache size alters. From the performance results, we can see that small size of data cache will cause less IPC performance. Because the Java benchmark programs are object-oriented, most memory accesses are concentrated in a limited area, this behavior will reduce memory cache miss, thus the performance effects for memory load miss is not prominent, such as in *Db, Javac, and Jess* benchmark programs. For those computation-intensive programs, such as *Compress, Mpeg, and Mtrt* (single-thread), they will access larger memory address ranges, the performance effects for memory load miss is bigger than the other programs.

Figure 7.1. IPC performances with different cache sizes

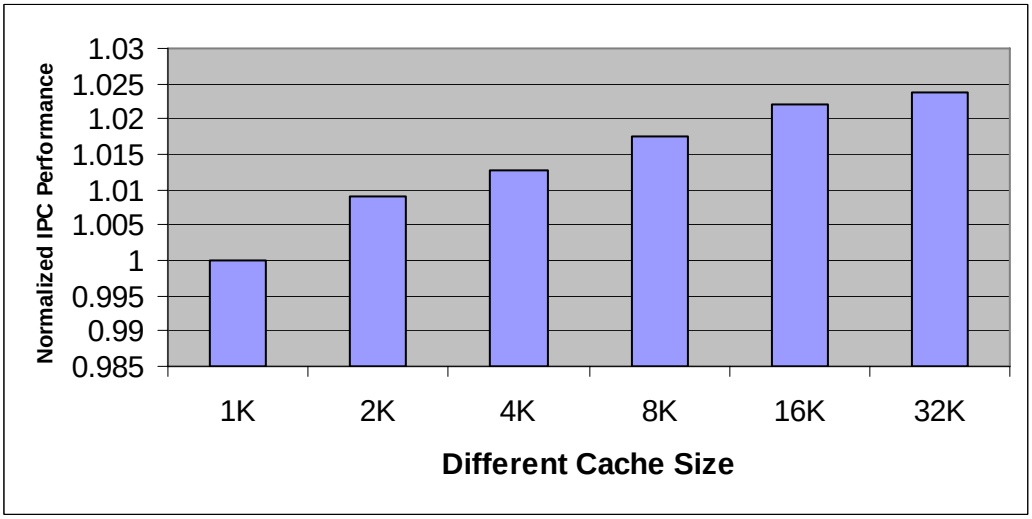
(a) Compress benchmark program



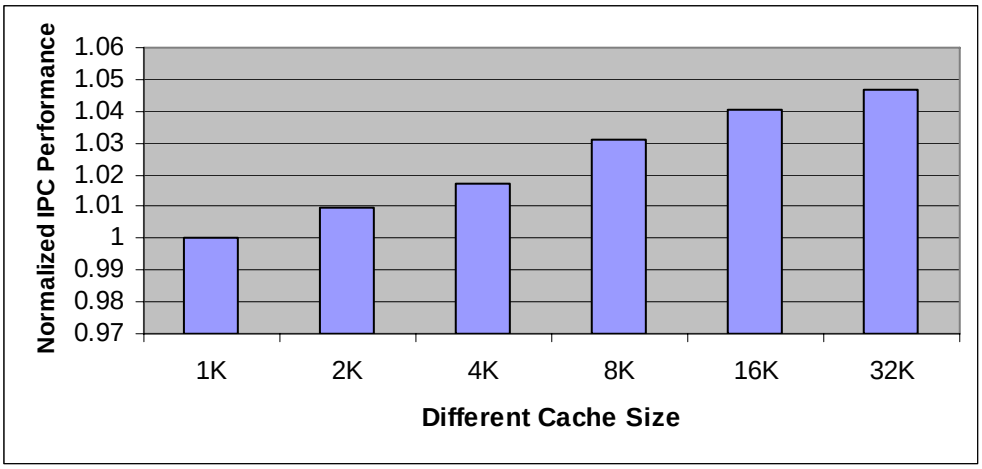
(b) Db benchmark program



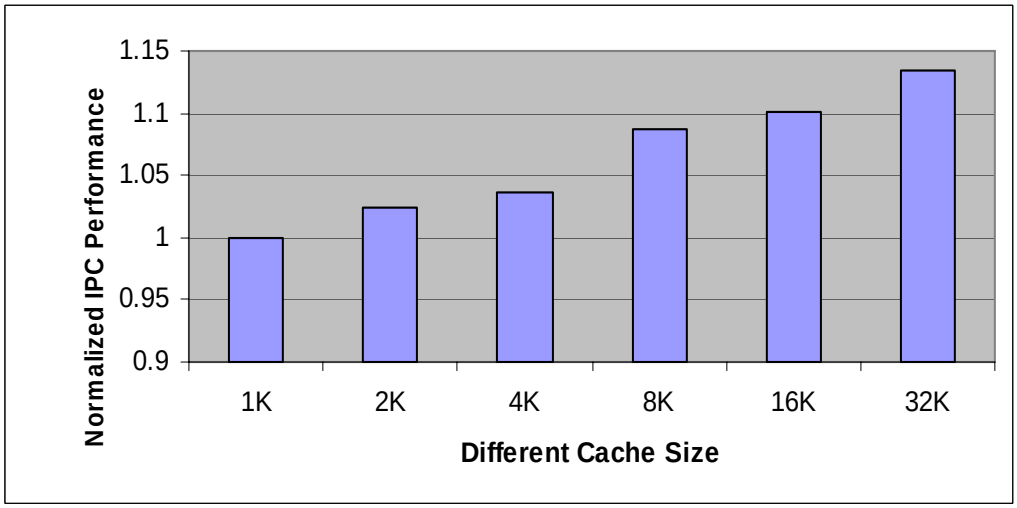
(c) Javac benchmark program



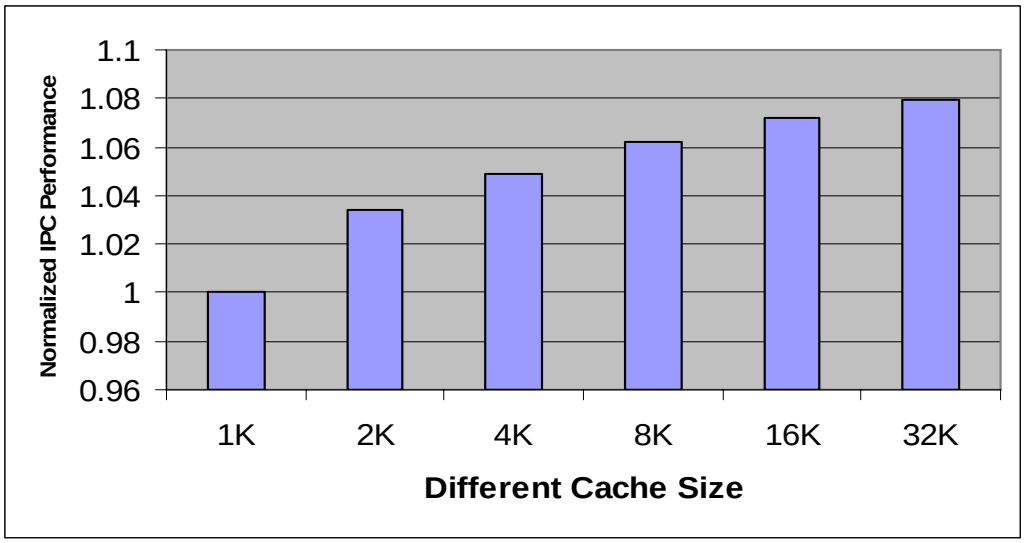
(d) Jess benchmark program



(e) Mpegaudio benchmark program



(f) Mtrt benchmark program



7.5.2.2 Cache Miss Rate vs. Cache Size

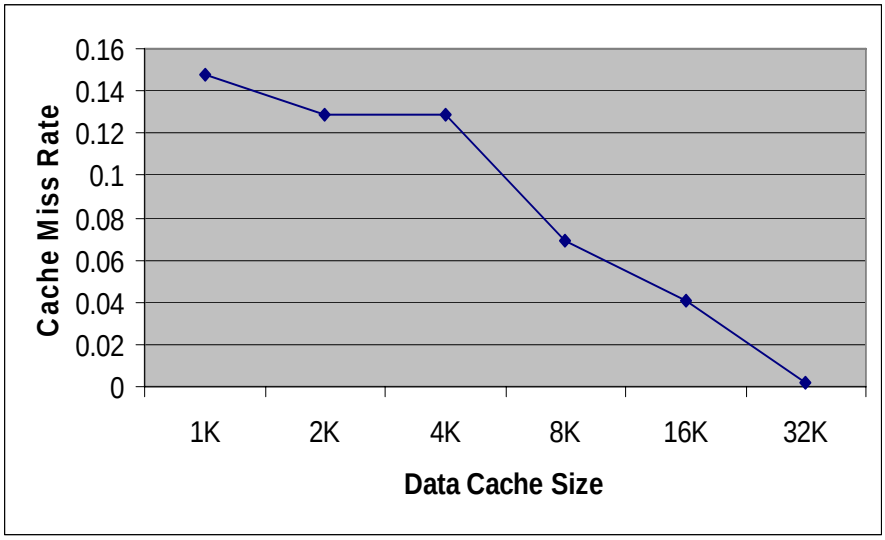
To further investigate the cache performance, we draw the figures with cache miss rate vs. cache size. We use these results to analyze how the cache size affects the cache miss rate for the benchmarks. All the results for the benchmarks are listed in Figure 7.2 (a) – (f).

The investigation of these figures shows that the cache miss rate will decrease as the data cache size increases, but different benchmark programs have different characteristics. For the *Compress* benchmark program, the cache miss rate will reduce a lot when the data cache size is larger than 4kbyte. For the benchmark programs of *Db, Javac and Jess,* the cache miss rates reduce nearly linearly with the cache sizes

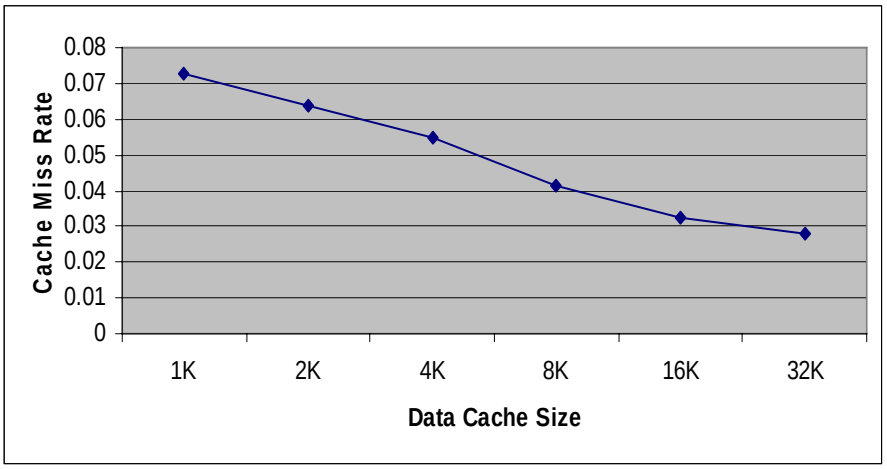
increasing. For the benchmark program -- Mpegaudio, its cache miss rate reduces slowly when the data cache size from 1Kbyte to 4Kbyte, later it reduces greatly when cache size is bigger than 4Kbyte. On the contrary, for the benchmark program -- *Mtrt*, its cache miss rate reduces a lot when the data cache size increase from 1Kbyte to 2Kbyte, later the cache miss rate reduces slowly as the data cache size increases.

Figure 7.2. Cache miss rate vs. cache size

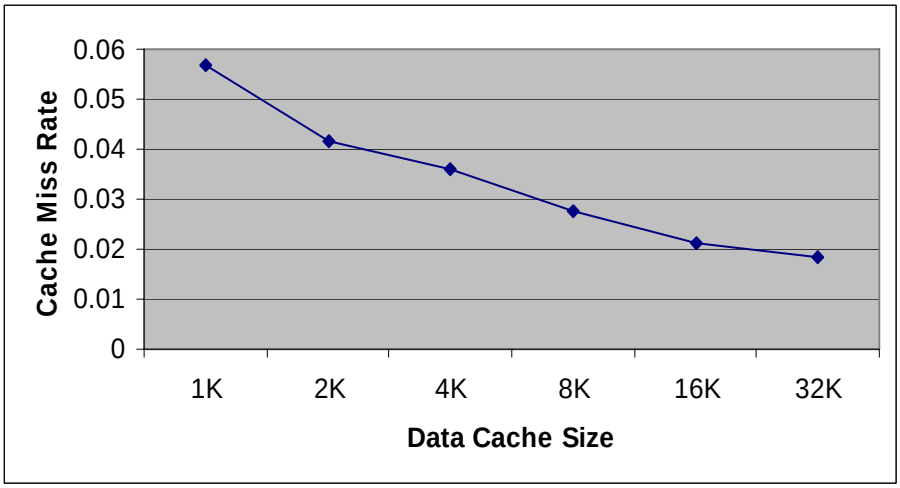
(a) Compress benchmark program



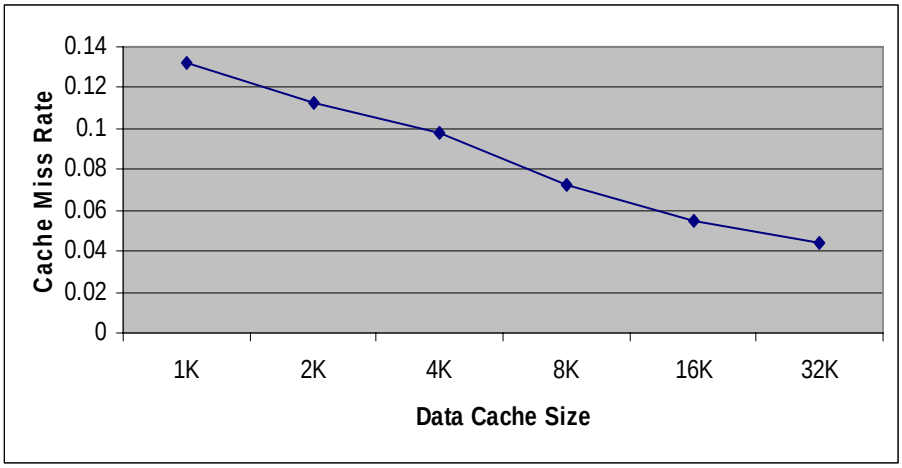
(b) Db benchmark program



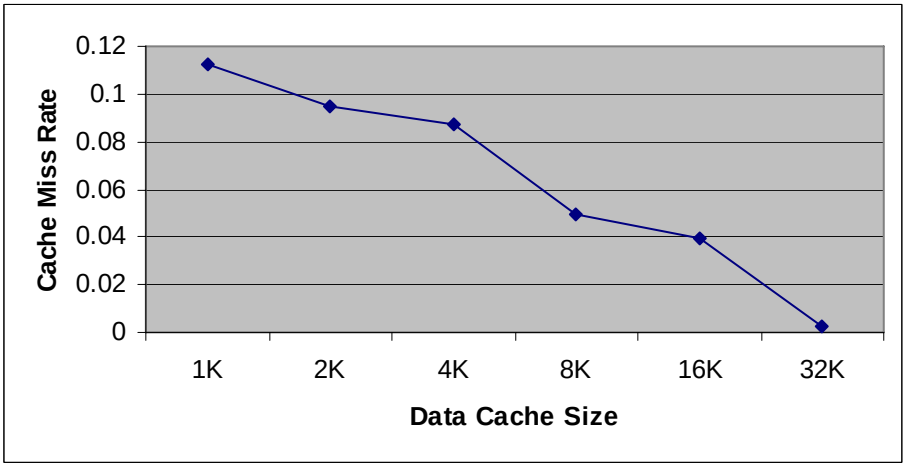
(c) Javac benchmark program



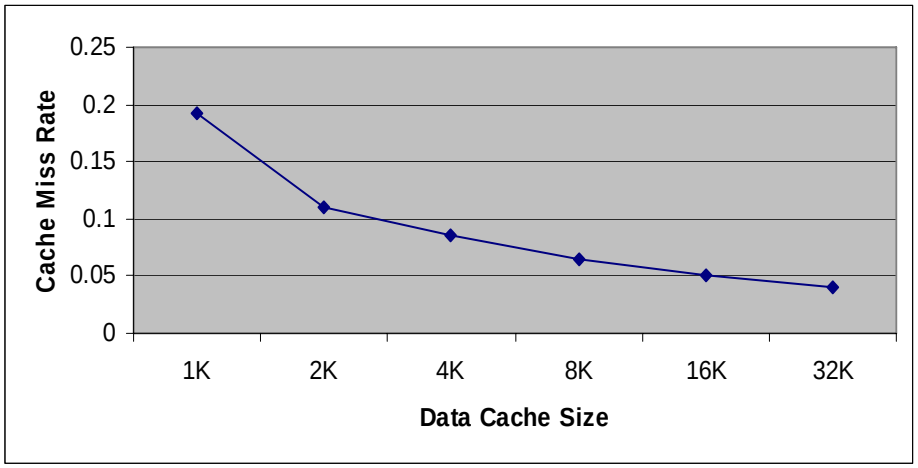
(d) Jess benchmark program



(e) Mpegaudio benchmark program



(f) Mtrt benchmark program



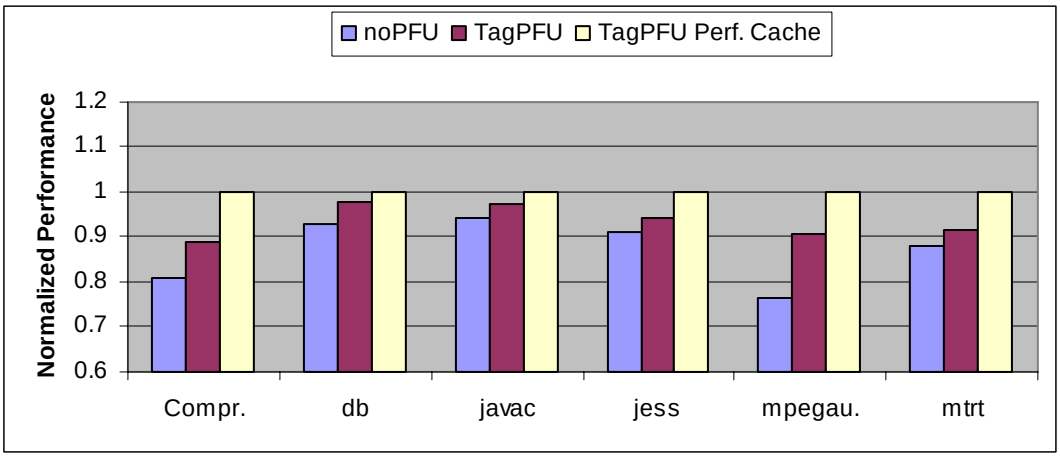
7.5.2.3 Performance Comparison using Different Scheduling Scheme

Figure 7.3 presents the normalized IPC performance results for TMSI processor in the three different scheduling scheme: the real cache (in the case of load cache miss, but without using tagPFU scheme), tagPFU scheme (with load cache miss) and tagPFU with perfect cache (in this case, we use tagPFU scheme on data cache, but no latency delay added) when the size of data cache are set at 1KB, 2KB, 4KB, 8KB, 16KB and 32KB respectively. In these three cases, the third case (tagPFU with perfect cache) is a theoretical upper-bound, which indicates the best performance when using tagPFU scheduling scheme. The experiment's results indicated that at the assumed cache configuration, tagPFU instruction schedule scheme can mitigate the performance degradation due to unpredicted memory load delay when the data cache size are set at

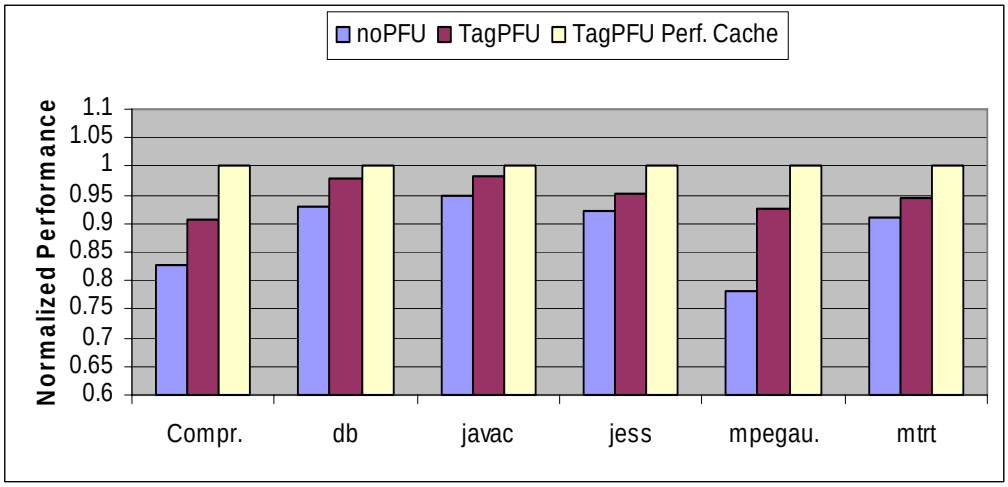
1KB, 2KB, 4KB, 8KB, 16KB and 32KB respectively. The results demonstrated that using tagPFU scheme can increase the IPC performance from 3% to 18%. In some conditions, some benchmark programs can achieve the ideal performance when using tagPFU scheme, for example, when the data cache size is assumed at 8Kbyte, the performance of the benchmark programs -- *Db and Javac* can nearly achieve the value with perfect cache when using tagPFU scheme. When the data cache size is assumed at 32Kbyte, the performance for the benchmark programs -- *Compress, Db, Javac, and Mpegaudio*, can achieve the value with perfect cache when using tagPFU scheme.

Figure 7.3. IPC performances with different scheduling scheme

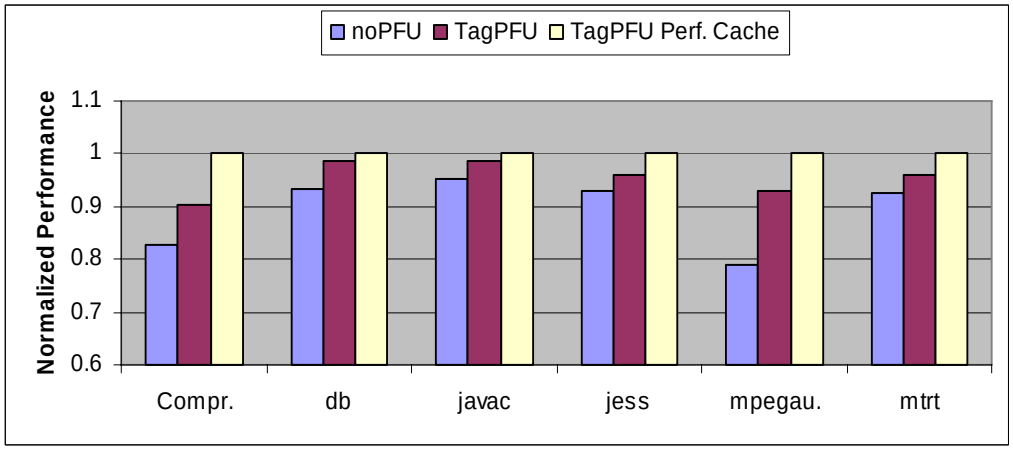
(a) Performance Improvement at 1KB data cache



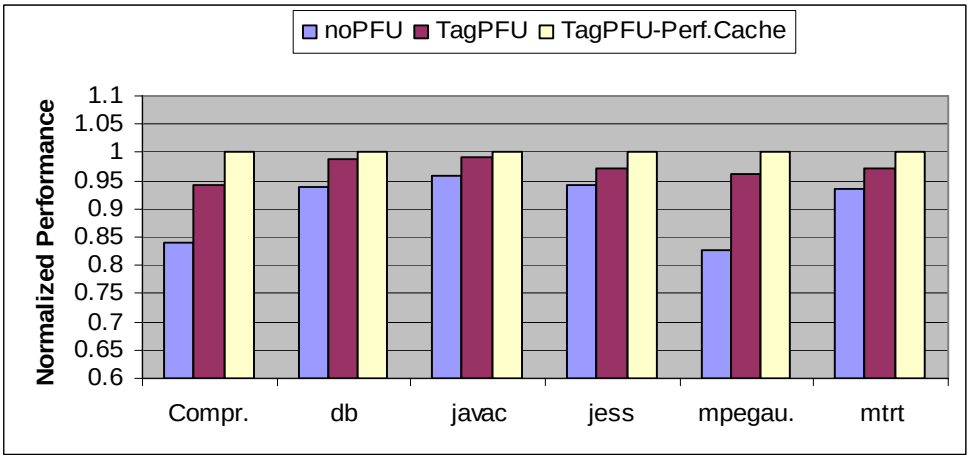
(b) Performance Improvement at 2KB data cache



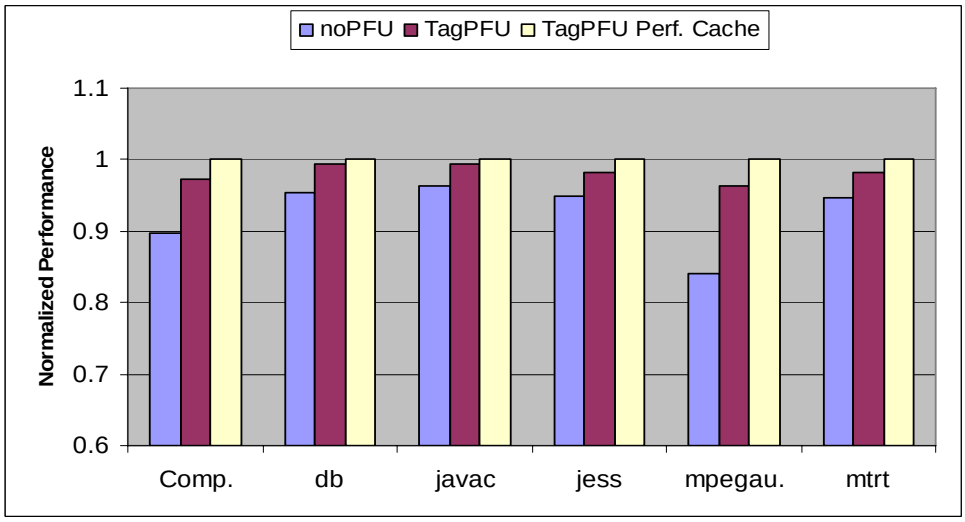
(c) Performance improvements at 4KB data cache



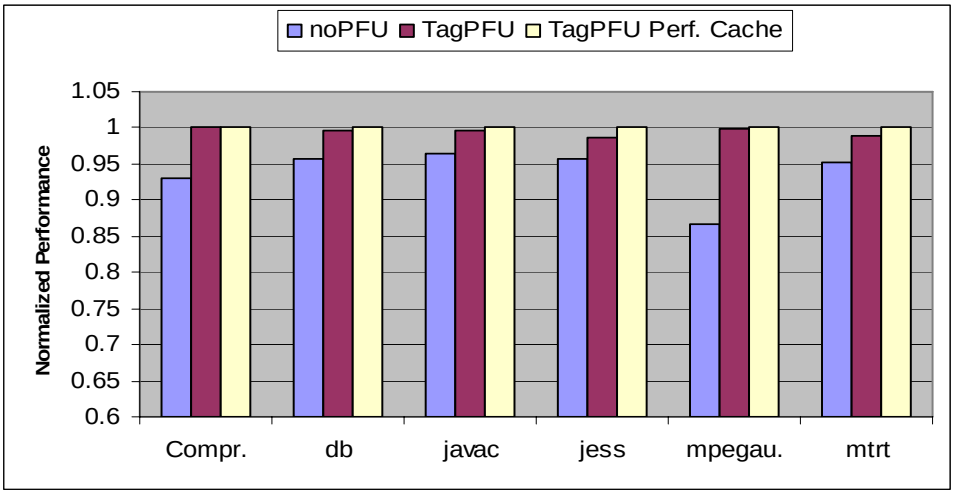
(d) Performance improvements at 8KB data cache



(e) Performance improvements at 16KB data cache



(f) Performance improvements at 32KB data cache



7.6 Conclusions

In this chapter, we exhibited two different CPU instruction execution models – out-of-order execution and in-order execution. Out-of-order execution model is used in major superscalar processors, whereas in-order execution model is used in VLIW / EPIC processors. The unpredictable memory load delay can be mitigated using dynamic scheduling techniques in out-of-order processors. But it cannot be mitigated in in-order processors, because they depend on compiler techniques to schedule instructions statically.

Several dynamic techniques used in VLIW / EPIC architecture to reduce the effects of long memory load delay are presented for comparison. To solve the unpredictable memory delay in TMSI Java ILP processor, we proposed a new implementation scheme – Tag-based PFU scheme. This scheme can reduce the effects of memory load delay and increase the performance as well. We also presented the performance results for the tag-based PFU scheme. The results show that the tag-based PFU scheme is effective.

Chapter 8
Conclusions

This chapter summarizes the thesis and discusses potential future development issues.

8.1 Conclusions

In the thesis, we proposed a General Tagged Execution Framework (GTEF). The conceptual framework employed a hardware abstract machine and caters for many existing pipelined computer architectures. To design a new processor with the proposed framework, we only need to design the specified tag-based abstract machine translator (TAMT) for the specified processors which will translate the instructions into a tag-based instruction format. This processor design methodology will be able to reduce the complexity of designing new processors, and reuse existent ILP hardware techniques.

The TAMT is the critical component in the framework, and may be viewed as a dynamic hardware translator or interpreter. This translator can translate RISC or CISC machine code into a universal RISC-like instruction format – tag-based RISC-like instruction format, and also can translate stack machine code, i.e. Java bytecode instruction, into tag-based RISC-like instruction format. The translation procedure

makes the executable internal instructions in both types of processors, whatever RISC or stack processors, be converted to a unique instruction format. This unique instruction format can easily be integrated with modern ILP execution hardware, superscalar or VLIW execution engine.

ILP is extensively used in modern high performance processors to achieve the performance. Register renaming is an important technique to increase ILP by removing false data dependencies dynamically. Register renaming technique is employed in TAMT to construct tag-based architecture. Therefore the tag-based instruction formats generated by TAMT will have removed data dependencies, and can be directly used by ILP execution hardware.

As stack-based processors have their specific features different with register-based RISC processors. In order to demonstrate how the GTEF scheme is applied in stack processors, we have fulfilled an implementation of a TAMT for a stack processor. The architecture of TAMT used in the stack processor can be incorporated with Tomasulo algorithm or other techniques to utilize modern ILP execution hardware to achieve high performance.

In TMAT used in stack processors, whenever a binary instruction enters the pipeline, the instruction is assigned a tag. With “mock” execution of the abstract machine, all

instructions are converted into tag-based instruction formats. The process of instruction tagging makes dataflow embedded in the stream of new-generated tag-based instruction. Thus dataflow techniques may be exploited to achieve out-of-order instruction execution and extract more ILP.

Based on the architecture of proposed stack-based TAMT in the thesis, we implemented a Java ILP processor as an example. By means of the proposed stack folding technique the Java ILP processor converted Java bytecode instructions to tag-based RISC-like instructions that are executed on a VLIW engine. The detailed design and implementation is depicted, and such related issues as stack folding, tag retention, speculation are also discussed in the thesis.

To demonstrate the effectiveness of the proposed Java ILP processor, we developed a trace-driven architectural simulator to verify the proposed architecture. The simulation results are encouraging when executing SpecJVM98 benchmark workload.

To tolerate unpredicted memory load delay in VLIW processors is a tough technical issue to improve the VLIW machine’s performance. The proposed Java ILP processor encounters the same problem due to the use of a VLIW in-order execution engine. To solve this problem, we proposed a modified tag-based PFU scheme based on the tag-based Java processor architecture. The simulation results demonstrate that the scheme

can not only alleviate the effects of data cache miss on IPC performance but also increase the performance.

8.2 Future Work

Given that the proposed abstract machine-based processor design framework is a conceptual framework, our future work will concentrate on realizing or building a real processor with this framework to justify the theoretical concept. Some issues that relate to the future work are discussed as follows.

8.2.1 SMT Architectural Support

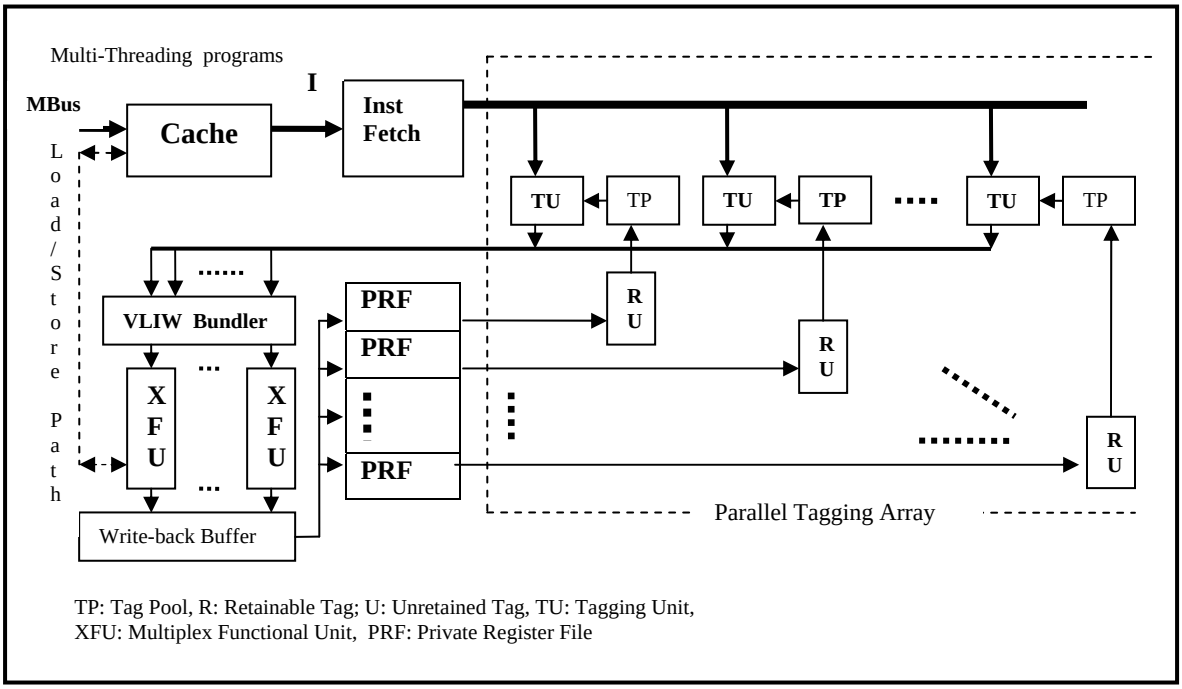
Simultaneous multithreading (SMT) is a variation on multithreading that combines hardware features of wide-issue SuperScalars with multi-threaded processors [34]. It can consumes both thread-level and instruction-level parallelism with greater instruction throughput and speedups. SMT processors have three advantages compared with superscalar processors. First, it does not need special hardware to schedule instructions from the different threads onto the functional units. Second, the resolution of the dependences can be handled by the dynamic scheduling capability. Third, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them.

There are three methods to implement multithreading on superscalar machines. They are coarse-grained multithreading, fine-grained multithreading and simultaneous

multithreading (SMT). In the coarse-grained multithreading machines, the long stalls can be partially hidden by switching to another thread that uses the resources of the processor. In the fine-grained multithreading, empty slots can be fully eliminated by the interleaving of threads. In the SMT case, TLP and ILP are exploited simultaneously with multiple threads using the issue slots within a single cycle.

The proposed tag-based processor architecture can be extended to support SMT in order to achieve higher speedups and throughput. To support SMT, we can provide multiple fetching units and tagging units (TU) with separate register file, program counter (PC) and a separate page table. To do in such way, multiple threads within SMT can share the common execution engine so that the high throughput can be achieved. In multi-threading supported Java ILP processor, bytecodes from different threads can be tagged by different tagging units and then bundled to the VLIW instruction to be executed in parallel, and the thread-level parallelism is achieved accordingly. Tagged instructions are from independent threads, they can be issued without regard to data dependences, but dependences within a thread will be handled by different TU. The schematic figure can be referred to the Figure 8.1**.**

Figure 8.1: The schematic for a SMT execution engine



In SMT machine, the memory can be shared by all threads through the virtual memory mechanisms, which already support multiprogramming. In proposed SMT architecture, multiple threads can share their common object or data via virtual memory system, therefore we should design a memory consistency model to guarantee the correctness of the program execution. When we execute Java programs on the proposed SMT architecture, the proposed memory consistency mechanism should respect the Java Memory Model (JMM) [35]. To meet this requirement, we can use sequential

consistency or release consistency memory model. Appropriate approaches need to be further investigated in our future research work.

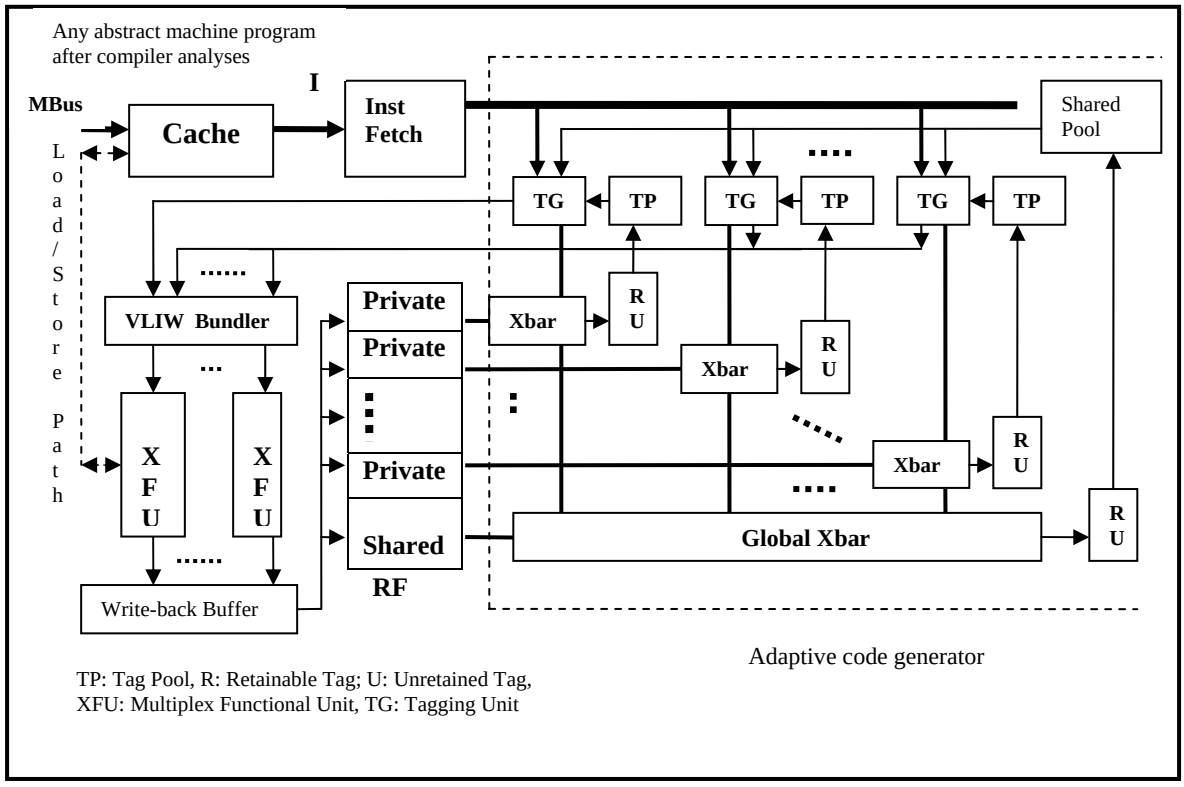
8.2.2 Scalability in Tag-based VLIW Architecture

To support large issue-window and higher issue rates in the proposed ILP processor register file will become a bottleneck as it is in traditional VLIW machines. To solve this problem, we have devised a scheme of multiple tagging units which uses register file partition. In this scheme, each tagging unit (TU) has its own private register file, and a common-used register file is provided to store global variables. This design takes advantage of the banked multi-ported register file architecture [37] to support multiple TUs with high performance. In this architecture the register bank will be partitioned to specific TUs, and a crossbar may be used to connect register banks with function units. This method will effectively reduce the pressure of the register file. We give a basic schematic framework to support multiple-tagging units in Figure 8.2.

As shown in Figure 8.2, Instruction Fetching Unit (IFU) will separate the instruction stream into independent code-segments, and then send them to individual TU. The instruction codes are pre-processed by the customized compiler, which can locate the independent code segment. The multiple tagging units tag instruction codes in parallel, and then send ready tagged instructions to VLIW bundler to build into VLIW instruction which will be issued to functional units. The instruction bundles execute in-order which makes issue logic simple. The execution results are tagged and communicated via a crossbar among tagging units. The tag will be set as retained when

it is needed by subsequent consumer or as un-retained which can be freed and reused by other instructions.

Figure 8.2: The schematic for a dynamic VLIW execution engine



8.2.3 Issues of Pipeline Efficiency

Most of the pipeline stages in a deeply pipelined, out-of-order superscalar processor are used for book-keeping tasks. As such, there is a good deal of inefficiencies. Many recent optimizations such as micro-op fusion essentially seek to reduce these

inefficiencies but internally having “complex” operations. It is a kind of partial reversal to CISC. Directly executing Java bytecode has a similar flavour. As a comparison, we may compile and execute Java benchmark – SpecJVM98 on a pure, register-based processor simulator and measure the ILP and instruction counts involved to compare the pipeline efficiency between the two techniques.

In order to execute SpecJVM98 benchmarks on a register-based processor simulator, we have alternative way to implement the task. We can choose a widely used superscalar performance simulator – SimpleScalar [23] as the simulation platform. Since SimpleScalar can not support to run JVM or Java programs. However, if we can directly compile Java programs into a register-based native binary format, then it can directly run on SimpleScalar. To do this, we can exploit the gcc-based static compiler for Java (gcj) to compile a set of standard Java benchmarks into static binary first, and then simulate these benchmarks using the SimpleScalar architecture simulator. Because SimpleScalar 3.0 only supports Alpha binary, or Portable Instruction Set Architecture (PISA) [23], if we can compile Java bytecode into Alpha static binary, we can use SimpleScalar to simulate Java benchmarks. This is a direct approach to executing Java benchmarks on register-based superscalar simulator. However, in order to generate Alpha machine binary code, we need a Compaq Alpha Tru64 computer. Currently we do not have this computer, therefore to conduct performance evaluation for SpecJVM98 benchmark programs with this approach will be one of our future works.

The other way to execute JVM and Java benchmark programs is to use Dynamic SimpleScalar (DSS) [109], which is an extension of SimpleScalar simulator. Although SimpleScalar did not support simulation of dynamic compilation, threads, or garbage collection, DSS can simulate Java programs running on a JVM, using just-in-time compilation, executing on a simulated multi-issue, out-of-order superscalar processor. Here we executed Java benchmarks on DSS simulator and obtained the following results show in Table 8.1.

Table 8.1. DSS simulation execution results

Benchmarks	**Simulation results**		
	Inst. counts (10^6)	Cycle counts(10^6)	ILP
Compress	2951	1733	1.7028
Db	2899	1702	1.7027
Jack	6741	3924	1.7177
Javac	6063	3540	1.7128
Jess	4871	2848	1.7102
Mpegaudio	3626	2129	1.7031
Mtrt	5046	2940	1.7165
Linpack	638	393	1.6219

In Table 8.1, we presented some execution results using DSS simulator. In these experiments, we run SpecJVM98 and Linpack benchmarks on DSS. We extracted instruction counts, cycle counts and obtained the ILP. From Table 8.1, we can see that when using a Just-in-Time compiling technique to execute Java programs on a modern RISC superscalar processor, the programs need execute much more times RISC instructions compared with the execution on a Java ILP processor. (The corresponding

Java instruction counts can be seen in Table 4.11.) The results demonstrate that if we use JIT technique to translate Java bytecode into RISC machine code to execute Java programs, a much higher overhead will be added. Thus, from the other point of view, it demonstrates that it is needed to build a high-performance Java processor for embedded system application.

Bibliography

1. A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. *Fast, Effective Code Generation in a Just-In-Time Java Compiler*. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, October 2000

2. A.F.de Souza and P.Rounce, *Dynamically Scheduling VLIW instructions*, Journal of Parallel and Distributed Computing, pp. 1480-1511, 2000

3. A. González, J. González, and M. Valero. *Virtual-Physical Registers.* In Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98) pp175-184, 1998.

4. A. Kim, M. Chang, *Advanced POC model-based Java instruction folding mechanism*, in: Proceedings of 26th EUROMICRO Conference, vol. 1, September 2000, pp.332–338.

5. A.V. Aho, R.Sthi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

6. A. Krall. *Efficient JavaVM Just-in-Time Compilation*, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp205, 1998

7. Amir Roth and Gurindar S.Sohi. *Speculative Data-Driven Multithreading.* Seventh International Symposium on High Performance Computer Architecture (HPCA-7), January 2001.

8. Antonio C.S.Beck, Luigi Carro. *A VLIW Low Power Java Processor for Embedded Applications.* In 17th Brazilian Symp. Integrated Circuit Design (SBCCI 2004), Sep.2003.

9. Arthur H. Veen. *Dataflow machine architecture*. ACM Computing Surveys, Vol. 18, Issue 4, December 1986.

10. A.R. Pleszkun and G.S.Sohi. *The Performance Potential of Multiple Functional Unit Processors.* In 15th Annual International Symposium on Computer Architecture, pages 37--44, May 1988.

11. Arvind, Rishiyur S. Nikhil. *Executing a Program on the MIT Tagged-Token Dataflow Architecture*. IEEE Trans. On Computers, Vol,39, No.3, March 1990.

12. Brad Calder and Dirk Grunwald. *Fast & Accurate Instruction Fetch and Branch Prediction.* Appear in 1994 Intl. Symp. On Computer Architecture, Chicago, April 1994.

13. B. Ramakrishna Rau. *Dynamic Scheduled VLIW Processors*. Proceedings of the 26th annual international symposium on Microarchitecture, Austin, Texas, United States, pp 80-92, 1993.

14. Brian Davis, Andrew Beatty, Kevin Casey, David Gregg and John Waldron. *The Case for Virtual Register machines*. ACM SIGPLAN Workshop: Interpreters, Virtual Machines and Emulators. IVME'03, June 2003, San Diego, USA.

15. B.S. Yang, S.M. Moon, S. Park, J.Lee, *LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation.* In the International Conference on Parallel Architectures and Compilation Techniques. October 1999.

16. Chris H. Perleberg and Alan Jay Smith. *Branch Target Buffer Design and Optimization.* IEEE Trans. On Computers, Vol. 42, No. 4, April 1993.

17. David J. Lijia. *Reducing the Branch Penalty in Pipelined Processors*. IEEE Computer, Vol. 21, Issue 7, pp.47-55, 1988.

18. David Landskov, Scott Davidson, and Bruce Shriver, *Local Microcode Compaction Techniques*, ACM Computing Surveys, Vol. 12, No. 3, September 1980.

19. David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, Wen-mei W. Hwu. *Dynamic Memory Disambiguation Using the Memory Conflict Buffer.* Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1994

20. D. Sima. *The Design Space of Register Renaming Techniques.* IEEE Micro, 20(5):70--83, Sept. 2000.

21. Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor.* Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.

22. David W. Wall. *Limits of Instruction-Level Parallelism*. Digital. Equipment Corporation. WRL Research Report 93/6

23. D.C. Burger and T.M. Austin. *The SimpleScalar tool set, version 2.0.* Computer Architecture News, 25(3):13—25, June, 1997

24. G. Hinton, D. Sager, M. Upton, D. Boggs, D. C. n, A. Kyker, and P. Roussel. *The microarchitecture of the Pentium 4 processor*. Intel Technical Journal, Q1 2001 Issue, Feb. 2001.

25. H. C. Wang, C. K. Yuen. *A General Framework to Build New CPUs by Mapping Abstract Machine Code to Instruction Level Parallel Execution Hardware.* ACM SIGARCH Computer Architecture News, Vol. 33, Issue 4, Nov. 2005, pp 113-120.

26. H. C. Wang, C. K. Yuen. *Exploiting Dataflow to Extract Java Instruction Level Parallelism on a Tag- based Multi-Issue Semi In-Order (TMSI) Processor.* IEEE International Parallel & Distributed Symposium 2006, Rhodes, Greece.

27. H. Dwyer, H.C. Torong. *An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts.* Proceedings of the 25th Annual International Symposium on Microarchitecture, pages 272--281, 1992.

28. Harlan McGhan and Mike O'Connor, *PicoJava: A Direct Execution Engine for Java Bytecode, Sun Microsystems,* IEEE Computer Magazine, 1998.

29. H. Sharangpani, and K. Arora. *Itanium Processor Microarchitecture.* IEEE Micro, vol. 20, iss. 5, Sept./Oct. 2000.

30. INMOS Limited, *Transputer Instruction Set – A Compiler Writer's Guide* , Prentice-Hall, London, 1988

31. Jan Edler, Mark D. Hill. http://www.cs.wisc.edu/~markhill/DineroIV

32. J. E. Smith, and G.S. Sohi, *The micro architecture of Superscalar Processors*, In proceedings of the IEEE, vol. 83, pp1609-1624, December 1995

33. J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. *Introducing the IA-64 Architecture*. IEEE Micro, 20(5):12--23, September /October 2000.

34. Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm, and Dean Tullsen. *Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading.* ACM Transactions on Computer Systems, August 1997.

35. Jeremy Manson, William Pugh and Sarita V.Adve. *The Java Memory Model.* In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), California, USA, January 12 -14, 2005.

36. Jeremy Manson and William Pugh. *Core Semantics of Multithreaded Java*. Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, Palo Alto, California, United States, Pages: 29 - 38, 2001.

37. Jessica H. Tseng, Krste Asanović. *Banked multiported register files for high-frequency superscalar microprocessors.* In Proceedings of the 30th annual international symposium on Computer architecture, San Diego, California, June 09 - 11, 2003, pp 62-71.

38. J.L. Bruno and T. Lassagne. *The Generation of Optimal Code for Stack Machines*. Journal of the Association for Computing Machinery, Vol. 22, No. 3, July 1975, pp. 382-396.

39. J. Michael O'Connor, Marc Tremblay, *PicoJava-I: The Java Virtual Machine in Hardware.* IEEE Micro, Vol. 17, Issue 2, pp 45-53, March 1997

40. John Glossner, et. al. *Delft-Java Link Translation Buffer*. In Proceedings of the 24th EUROMICRO conference, Vol.1, pages 221–228, Vasteras, Sweden, August. 25-27, 1998.

41. John L Hennessy and David A Patterson, *Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, Inc*., 1996.

42. T. Shpeisman and M. Tikir. *Generating Efficient Stack Code for Java*. Technical report, University of Maryland, 1999.

43. Kenneth C. Yeager. *The MIPS R10000 Superscalar Microprocessor.* IEEE Micro April 1996 (Vol. 16, No. 2) pp. 28-40

44. Kevin Scott and Kevin Skadron. *BLP: Applying ILP Techniques to Bytecode Execution*. Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, Sept 17, 2000.

45. Krishna M. Kavi, Roberto Giorgi and Joseph Arul, *Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation*. IEEE Trans. On Computers, VOL 50, No. 8, August 2001

46. K. Ebcio˘glu, E. Altman, and E. Hokenek. *A Java ILP machine based on fast dynamic compilation.* In MASCOTS'97, - International Workshop on Security and Efficiency Aspects of Java, 1997.

47. K. Ebcioğlu , Erik R. Altman, *DAISY: dynamic compilation for 100% architectural compatibility*, ACM SIGARCH Computer Architecture News, v.25 n.2, p.26-37, May 1997

48. L.C.Chang, L.R.Ton, M.F. Kao, and C.P.Chung. *Stack operations folding in Java Processors*. IEE Proc. Comput. Digital Technology, Vol. 145, No 5, Sept. 1998

49. Lee, J. and Smith, A.J. *Branch prediction strategies and branch target buffer design*. IEEE Computer, Jan. 1984, pages 6-22.

50. Lee-Ren Ton, Lung-Chung Chang, Chung-Ping Chung. *An analytical POC stack operations folding for continuous and discontinuous Java bytecodes.* Journal of Systems Architecture 48 (20020 pp. 1--16

51. L.Gwennap. *Intel's Uses Decoupled Superscalar Design.* Microprocessor Report, pp. 9-15, Feb. 1995.

52. Linpack, http://www.netlib.org/linpack

53. L.R. Ton, Lung-Chung Chang, Min-Fu Kao, Han-Min Tseng, *Instruction Folding in Java Processor*, the International Conference on Parallel and Distributed Systems, 1997

54. L.R. Ton, L.C. Chang, C.P. Chung. *Exploiting Java bytecode parallelism by dynamic folding model.* Proceedings of the 6th International Euro-Par Parallel Processing Conference Lecture Notes in Computer Science, Vol. 1900, August 2000, pp. 994-997

55. Lori Carter, Weihaw Chuang and Brad Calder. *An EPIC Processor with Pending Functional Units.* In Proceedings of the 4th International Symposium on High Performance Computing (ISHPC), May 2002, Springer-Verlag.

56. Machael D. Smith, Mark Horowitz, Monica S.Lam. *Efficient Superscalar Performance Through Boosting.* Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA, Oct. 1992.

57. M. Anton Ertl. *Stack Caching for Interpreters*. ACM SIGPLAN'95, Conference on Proramming Language Design and Implementation, pages 315-327, 1995.

58. Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In Proceedings of the 26th International Symposium on Microarchitecture (MICRO 26), P202-213, Dec. 1993, Austin, Texas, USA.

59. Mihai Budiu, Pedro V. Artigas and Seth Copen Goldstein. Dataflow: A complement to Superscalar. Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on (2005), pp. 177-186.

60. Mark D. Hill and Alan Jay Smith, *Experimental Evaluation of On-Chip Microprocessor Cache Memories.* Proc. Eleventh International Symposium on Computer Architecture, June 1984, Ann Arbor, MI. (Dinero IV)

61. M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and Wen-mei W. Hwu, *An Architectural Framework for Run-Time Optimization* . IEEE Transactions on Computers, Vol. 50, No. 6, June 2001, pp. 567-589.

62. M.G. Burke, J.D.Choi, S.Fink, D.Grove, M. Hind, V. Sarkar, M.J. Serrano, V.Sreedhar, H. Srinivasan, and J. Whaley, *The Jalapeno dynamic optimizing compiler for Java*, In Proceedings ACM 1999 Java Grande Conference, 1999, pp.129-141.

63. Michael G., Erik R. Altman, S. Sathaye, Paul Ledak, and David Appenzeller. *Dynamic and Transparent Binary Translation,* IEEE Computer, March 2000, pp. 54~59.

64. Mike Johnson. *Superscalar Microprocessor design*. Prentice Hall Series, 1991

65. Michael K. Chen and Hunle Olukotun. The *Jrpm System for Dynamically parallelizing Java Programs.* Proceedings of ISCA-30. June 2003, San Diego, CA, USA.

66. Martin Maierhofer, and M. Anton Ertl. *Optimizing Stack Code.* Forth-Tagung 1997, Ludwigshafen.

67. M. Lam. *Software pipelining: An effective scheduling technique for VLIW machines*. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 318-328. Published as *SIGPLAN Notices 23* (7), July 1988.

68. Michael S. Schlansker, B.Ramakrishna Rau. *EPIC: An Architecture for Instruction-Level Parallel Processors.* HP lab Technical Report 1999.

69. M.Tremblay, J.Chan, S. Chaudhry, Andrew W. Conigliaro, S.S.Tse. *The MAJC Architecture: A Synthesis of Parallelism and Scalability.* IEEE Micro Vol. 20, (6), Nov. 2000, pp. 12 -25.

70. M.W. El-kharashi, F. Elguibaly, K.F.Li. *A robust stack folding approach for Java processors: an operand extraction-based algorithm.* Journal of Systems Architecture 47 (2001) p.697-726

71. M. W. El-Kharashi, Fayez Elguibaly, Kin F. Li. *Adapting Tomasulo's Algorithm for Bytecode Folding Based Java Processors.* ACM Computer Architecture News, pp. 1–8, Dec. 2001.

72. Norman P.Jouppi. *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*. ACM SIGPLAN Notices. 1989.

73. N.VijayKrishnan, N. ranganathan, R. Gadekarla, *Object-oriented architectural support for a Java processor.* ECOOP'98, the 12th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Springer, New York, NY, vol. 1445, 1998, pp. 330-354.

74. N. VijayKrishnan, *Issues in the Design of a JAVA Processor Architecture.* PhD dissertation, University of South Florida, Tampa, FL-33620. December 1998.

75. Perry H.Wang, et al, *Memory Latency-Tolerance Approaches for Itanium Processors Out-of-Order Execution vs. Speculative Pre-computation*. Proceeding of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02). Page 187.

76. Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. *Data-Driven and Demand-Driven Computer Architecture*. ACM Computing Surveys, Vol. 14, Issue 1, March 1982.

77. Philip J. Koopman, Jr. *Stack Computers: the new wave*. 1989

78. Philip J. Koopman, Jr. *A Preliminary Exploration of Optimized Stack Code Generation.* Journal of Forth Applications and Research, 1994, 6(3) pp. 241-251.

79. R. Achutharaman, R. Govindarajan, G. Hariprakash, Amos R. Omondi. *Exploiting Java-ILP on a Simultaneous Multi-Trace Instruction Issue (SMTI) Processor*, International Parallel and Distributed Processing Symposium, pp.76a, 2003.

80. R.A. Iannucci. *Toward A dataflow / Von Neumann Hybrid Architecture.* In 15th Annual International Symposium on Computer Architecture, pages 131--140, June 1988.

81. R.E. Kessler. *The Alpha 21264 Microprocessor.* IEEE Micro［C］. Haifa:IEEE, 1999,19(2):24-36.

82. R. Helaihel, and K. Olukotun, *JMTP: An Architecture for Exploiting Concurrency in Embedded Java Applicatins with Real-time Considerations*. In the international conference on Computer-Aided Design, Nov. 1999, pp. 551-557

83. Rahul Kapoor, Subramanya Sastry, Craig Zilles. *Stack Renaming of the Java Virtual Machine (1996).* http://citeseer.ist.psu.edu/kapoor96stack.html

84. R.M. Keller. *Look-ahead processors*. Computing Surveys, 7(4): 177-195, December, 1975

85. R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

86. R. P. Colwell, Robert.P.NIX, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. *A VLIW Architecture for a Trace Scheduling Compiler.* IEEE Trans. On Computers, Vol. 37, No.8, August 1988.

87. R. Radhakrishnan, N.Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural issues in Java runtime systems," Tech. Rep. TR-990719, 1999.

88. R.Radhakrishnan, Deependra Talla and Lizy Kurian John, *Allowing for ILP in an Embedded Java Processor.* In Proceedings of the 27th International Symposium on Computer Architecture, pages 294--305, June 2000.

89. R. Radhakrishnan, Deependra Talla and L. K .John, *Characterization of Java application at Bytecode and Ultra-SPARC machine code level*, In Proceedings of IEEE International Conference on Computer Design (Austin, TX, October 1999), pp. 281--284.

90. R.Radhakrishnan, N.Vijaykrishnan, L.K.John, A.Sivasubhramaniam, J.Rubio, Sabharinathan, *Java Runtime Systems: Characterization and Architectural implementation*. IEEE Trans. On Computers, Vol. 50, No. 2, Feb. 2001

91. R. Vall´ee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, Vijay Sundaresan. *Soot - a Java Bytecode Optimization Framework*. http://www.sable.mcgill.ca/soot/.

92. Stephan Diehl, P.Hartel, P.Sestoft, *Abstract machines for programming language implementation.* Future Generation Computer Systems, Vol. 16 (2000), pp 739--751

93. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/

94. S.P. Song. IBM's Power3 to Replace P2SC. Microprocessor Report, Micro Design Resources, Vol. 11, No. 15, 1997, pp. 23—27

95. S.S REDDI and E.A. FEUSTEL. *A Concept Framework for Computer Architecture.* ACM Computing Surveys, Vol. 8, No.2, June 1976.

96. S. T. Srinivasan and Alvin R. Lebeck. *Load Latency Tolerance In Dynamically Scheduled Processors.* Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture. Dallas, Texas, United States, pp 148-159, 1998.

97. Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. *Simultaneous Multithreading: A Platform for Next-generation Processors*. *IEEE Micro*, September/October 1997.

98. Sudheendra Hangal and Mike O'Connor. *Performance analysis and validation of the PicoJava processor.* IEEE Micro, 1999

99. Sun Microsystems Inc., *PicoJava-II Micro architecture Guide*, Sun Microsystems, CA, USA, March 1999

100. Takashi Aoki. *On the Software Virtual Machine for the Real Hardware Stack Machine,* Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01). Monterey, California, USA April 23–24, 2001

101. T. Hara and H. Ando, *Performance comparison of ILP machines with cycle time evaluation.* Proc. of the 23rd Annual International Symposium on Computer Architecture, pp. 213~224, March 1996.

102. "The Kaffe Virtual Machine", http://www.kaffe.org

103. The Microengine Company, Newport Beach,California, USA, Pascal Microengine Computer User's Manual, 1979.

104. T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA, 1996

105. Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. *Optimization of Instruction Fetch Mechanisms for High Issue Rates*. Proceedings of the

22nd Annual International Symposium on Computer Architecture (Santa Margherita, Italy), June. 1995.

106. Wei-Chung Hsu, Charles N. Fischer, and James R. GoodMan. *On the Minimization of Loads / Stores in Local Register Allocation*. IEEE Trans. On Software Engineering, Vol. 15, No. 10, Oct. 1989.

107. Wen-mei Hwu and Yale N. Patt. *HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality*. Proceedings of the 13th annual international symposium on Computer architecture, 1986, Tokyo, Japan, pp 297-306.

108. Yamin Li, San Li, Xianzhu Wang, and Wanming Chu. *JAViR—Exploiting Instruction Level Parallelism for JAVA Machine by Using Virtual Registers*. The 2nd European IASTED Inter. Conf. on Parallel and Distributed Systems, July, 1998 Vienna, Austria.

109. Xianglong Huang, J.Eliot B. Moss, Kathryn S. McKinley, Steve Blackburn, and Doug Burger. Dynamic SimpleScalar: Simulating Java Virtual Machines. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-03-03. February 2003.

110. The Arm White paper. “High performance Java on embedded devices – Jazelle technology: ARM acceleration technology for the Java Platform”. Arm Ltd September 2004.

111. Espresso, http://vodka.auroravlsi.com

112. Lightfoot Java CPU, www.dctl.com

113. Jstar, www.nazomi.com