Speculative Loop-Pipelining in Binary Translation for Hardware Acceleration

Sejong Oh, Tag Gon Kim, *Senior Member, IEEE*, Jeonghun Cho, *Member, IEEE*, and Elaheh Bozorgzadeh, *Member, IEEE*

***Abstract*—Multimedia and DSP applications have several computationally intensive kernels which are often offloaded and accelerated by application-specific hardware. This paper presents a speculative loop pipelining technique to overcome limitations of binary translation for hardware acceleration. Although many compilers have been developed at source level, it is desirable to translate the binary targeted to popular processors onto hardware for several practical benefits. However, the translated code can be less optimized. In particular, it is difficult to optimize memory accesses on binary to exploit pipeline parallelism since memory optimization techniques require perfect dependence information for correctness and efficiency. This information is not often available at binary level or even at the source level. Our technique synthesizes the pipeline with memory dependence speculation and postpones some phases of compilation by generating a small dependence analysis code or logic which makes use of runtime values. Such speculative optimization achieves the large amount of parallelism and does not depend on any user annotation. The experimental results show a promising speedup of up to 2.53 compared with the code in which memory accesses are not optimized in the pipeline fashion due to conservative memory analysis. In addition, we have evaluated our technique at hardware level implementation on FPGA devices and achieved comparable clock frequency and power consumption compared to a conservative method while achieving significant improvement in throughput.**

***Index Terms*—Binary translation, loop pipelining, memory optimization, reconfigurable computing.**

I. Introduction

BINARY translation has been studied in general-purpose architectures to run legacy software applications transparently or to virtualize underlying hardware tools [1]. In particular, many binary translators have been developed to run legacy software applications on new architectures for commercial reasons since only the binary is available in many cases and software migration usually costs significant time and effort.

Multimedia and DSP applications have several computationally intensive kernels which are often offloaded and accelerated by application-specific hardware. When binary translation is applied to hardware acceleration, the compilation flow is different from conventional binary translators. This is because the target architecture contains an application-specific hardware as well as a conventional embedded processor as a host processor. Therefore, the binary translator for hardware acceleration translates critical kernels of the software to the accelerator such as very long instruction word (VLIW) DSP, coarse-grained reconfigurable architecture (CRA), fine-grained reconfigurable architecture (FRA), and custom logic for application-specified integrated circuit (ASIC) and application-specific instruction-set processor (ASIP) [2], [3].

For hardware acceleration, binary translation is not commonly used. Conventional design tools either work on high-level source codes or require the developer to write the netlist for high-performance design. Binary translation is attractive for various practical and commercial reasons. First, different architectures have different design flows even though they fall in the same type of accelerator. For example, design flows and tools for reconfigurable architectures are different based on vendors and devices. Second, company's existing tools may not adopt a new design flow [3]. In addition, migration of application can be a very time-consuming task for application developers. Even some compilers use dialects of standard language or their own annotation [4], [5]. Finally, many critical kernels are often written in assembly.

One of the drawbacks in binary translation is that the code can be less optimized due to the lack of high-level information. In particular, memory access optimization for hardware synthesis is the key to achieve a large amount of parallelism, but the loss of high-level information usually results in inefficient pipelines. In multimedia applications, the critical kernels for stream-based computations consist of several loops which read data from input streams and write output data to output streams or scalar variables over every iteration. High-level compilers analyze memory dependence between memory accesses and generate code or netlist in the pipeline fashion. To analyze memory dependences, high-level compilers make use of an information supported by high-level languages and user's annotation, such as array, type, control constructors, function call, domain-specific operators, and keywords.

It is difficult to analyze memory dependence statically at binary level because binaries do not have high-level information. Although memory dependence can be discovered by sophisticated interprocedural analyzers in some cases, some values, such as the size of the image and the address of dynamically allocated object, are specified at runtime. In addition, if the binary is a library or a dynamic module, interprocedural analysis will not work because the analysis tool cannot

Manuscript received March 12, 2007; revised June 20, 2007. This work was supported in part by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2006-331-D00449). This paper was recommended by Associate Editor R. Camposano.

S. Oh and T. G. Kim are with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, Korea.

J. Cho is with the School of Electrical Engineering and Computer Science, Kyungpook National University, Taegu 702-701, Korea.

E. Bozorgzadeh is with the Department of Computer Science, University of California, Irvine, CA 92697 USA.

Digital Object Identifier 10.1109/TCAD.2008.915533

```
void fir(int* in, int* out, int len) {
   for(i=0; i< len; i++) {
      *o = c0 * in[0] + c1 * in[1] + c2 * in[2]
         + c3 * in[3] + c4 * in[4];
      in++; o++;
   }
}
```

Fig. 1. FIR filter example.

analyze the procedures not included in the library. Therefore, even though many streams have no memory dependence in real applications, binary translators should be conservative in order to maintain correctness, which makes the produced code unpipelined.

In this paper, we propose a speculative loop pipelining with runtime dependence tests in order to achieve loop pipelining and overcome the limitation of dependence analysis in the binary translation. This technique generates the netlist speculatively at compile time. Then, it modifies the statically produced netlist according to the result of runtime analysis. The earlier version of this paper was presented in ICCAD 2005 [6]. This paper significantly improves the previous version as follows.

1) It includes the detailed algorithm description to recover memory access pattern with runtime values from a binary.
2) It incorporates a data-reuse technique to decrease the number of memory accesses at runtime.
3) It presents two additional dependence inspection techniques with significant theoretical extension of generating dependence inspectors.
4) It extends the experimental framework in such a way that simulation modules, such as an ARM processor, a reconfigurable processor, and buses, are cycle-accurate in order to accurately measure the performance.
5) It shows the performance evaluation and analysis with additional benchmark programs.
6) It includes hardware implementation on FPGA device and reports the impact at hardware level in terms of area, clock frequency, and power consumption.

II. Motivation

In this section, we introduce our motivation and illustrate the speculative loop pipelining technique through a simple example. Fig. 1 shows a finite-impulse response (FIR) filter code written in C. This code has two streams that are independent during the loop execution. After the FIR code is compiled into a binary, the binary translator recovers the data flow from the compiled binary using decompilation techniques, as shown in Fig. 2(a). This code has pointer variables, in and out, pointing out to dynamically allocated objects. The size of data, len, is not known at compile time. Because of conservative dependence analysis, memory accesses for the two streams are connected with dependence edges, which are shown as dashed lines in Fig. 2(a). Those edges cause memory dependence across loop iterations, and the pipeline must stall until the store operation notifies the load operations that its operation is done. In contrast, when dependence edges between the load and store operations are eliminated, the load operations feed the pipeline without waiting for the completion of the store operations.

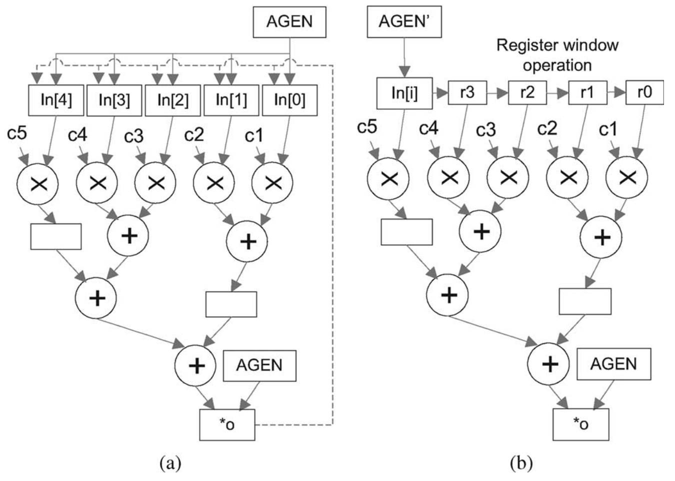


Fig. 2. Comparison between the data flow of the FIR code with conservative memory analysis and the one with accurate memory analysis. (a) Data flow with loop-carried dependence. (b) Pipelined data flow with data reuse.

Furthermore, we apply register promotion technique [7] in order to eliminate redundant memory accesses with data reuse. Fig. 2(b) shows the optimized data flow by the data-reuse technique. Instead of five load operations at every iteration, the data flow reuses loaded data at previous iterations. Accordingly, the number of dynamic memory loads is decreased from $5 \times (\text{len})$ to $5 + (\text{len} - 1)$. Since the performance of the data flow is usually limited by memory bandwidth, data reuse improves performance significantly. This optimization can be applied when the input stream is not dependent any other store operations. Significant improvement cannot be obtained with a conservative memory analysis in binary translation.

In speculative loop pipelining, we first generate the most optimistic netlist like the netlist shown in Fig. 2(b) with a small amount of code or logic called *dependence inspector* for runtime dependence tests. This code runs before or during the execution of the loop code. If memory dependence is discovered at runtime, the host processor stops the accelerator and rolls back the accelerator to the conservative mode which has the correct memory synchronization [see Fig. 2(a)]. Then, the host processor reexecutes the accelerator. Once the pipeline is reconfigured to the conservative mode, it is not reconfigured to the speculative mode again. In many applications in multimedia domain, memory dependences of the streams do not change dynamically. If memory accesses are discovered to be dependent at runtime, they are likely to be dependent until the end of the program execution. Therefore, it is not necessary to turn back to the speculative mode. For example, memory dependence specifiers, such as restrict keyword in C99 and other pointer-alias specifiers, specify dependence statically in high-level languages. In such applications, the speculation is highly acceptable since memory access streams in the loop do not have dependence in many cases.

This speculative approach is useful only when the overhead of the runtime dependence test can be recovered by the execution of optimized code. In this paper, the dependence test consists of a few simple inequality equations to compare the access patterns of memory regions in the loop. This test can be realized as a small software code executed before or along with the loop execution, or through hardware implementation with a small amount of logic. If the test runs on the host processor before the loop execution, the overhead is recovered by the pipelined loop execution. However, when the number of loop execution is too small to recover the overhead, the time overhead can be reduced or eliminated by implementing the test onto the accelerator. Section VI-C gives the details of the runtime dependence test.

III. Target Architecture

A. Architectural Assumptions

We assume that the target architecture consists of a general-purpose processor and an accelerator, sharing the same data memory space [8]–[11]. A reduced instruction set computer (RISC) processor is usually used as a general-purpose processor to run control-intensive parts of the application. As an accelerator, VLIW-based DSPs, FRAs, CRAs, and application-specific circuits for ASICs and ASIPs have been studied and used in order to speed up computationally intensive parts of the application with high degree of pipeline and parallelism.

When the speculation fails, the loop needs to be executed according to the correct runtime order of memory accesses. In order to roll back to the correct execution, we need the following mechanisms.

1) A rollback mechanism to the correct code: The accelerator can run rollback to the correct code to execute memory accesses in the correct runtime order.
2) A mechanism of preserving state: Data memory must not be altered by speculative memory writes for the possible reexecution of the loop.
3) A mechanism of detecting the miss-speculation: It must be possible to generate efficient dependence inspectors.

Rollback to the correct code is realized differently for different architectures. For VLIW-based architectures, the rollback is easily realized by means of multiversioning the loop. For example, the compiler generates both the speculatively pipelined code and the unpipelined code. If the miss-speculation is detected by runtime analysis, the unpipelined code is executed, instead of the speculative pipelined code. A similar approach can be applied to CRAs since modern CRAs need a much smaller amount of configuration bits compared to FRAs, and they usually provide partial and dynamic reconfiguration [8], [11]. Multiple configurations for different patterns of memory dependence can be prepared, and the correct one can be loaded into the device when the speculation fails.

Rollback is not trivial for FRAs, such as FPGAs. The loop is usually synthesized by traditional synchronous loop pipelining techniques. Therefore, a different result of memory dependence analysis may result in a completely different pipeline, because scheduling and sharing can be significantly affected. However, it is unacceptable with FRAs to redo synthesis and place-and-route at runtime or to prepare multiple versions of the pipeline for the different analysis results because of long configuration time and large memory space for configuration bits. To address this problem, we present a speculative loop pipelining approach whose memory synchronization is based on a handshaking

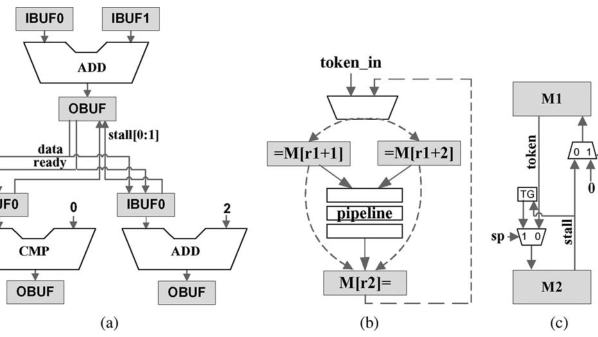


Fig. 3. Handshaking communication and token network for memory synchronization. TG generates a token for $M2$ in the speculative mode.

token network [12]. We describe the details in Section III-B. For application-specific circuits, the loop and token network can be synthesized in the same way as FPGAs.

There are two methods to preserve state in data memory: 1) preventing speculative memory writes and 2) maintaining a copy of memory regions accessed by speculative memory writes. The second method is attractive when the data are fetched from a slow external memory into a fast local memory inside the accelerator, because it reduces the number of external memory accesses. In this paper, we do not assume any distinct memory, so that all speculative writes are prevented.

In the rest of this paper, we select FPGAs as a hardware accelerator and focus on synthesizing the speculatively pipelined loops onto FPGAs with dependence inspectors.

B. Synthesis of the Speculatively Pipelined Loop Onto FPGA

The synthesis method in this paper is based on a handshaking-based approach with dynamic memory synchronization, which is similar to compiler for application-specific hardware (CASH) [12], Xilinx's CHiMPS [13], and XPP [11]. The pipeline architecture is composed of a set of predefined hardware operators, and they are connected through a handshaking communication network. Each predefined operator is a word-width operator composed of an arithmetic logic unit (ALU), a finite-state machine for the handshaking protocol, and buffers. Fig. 3(a) shows an example of three connected modules and signals for the handshaking protocol. The output buffer has a single register, data signals, and a ready signal. The input buffer has a single register (or a set of registers) to build a first-in-first-out (FIFO) and assert its stall signal when the buffer is full.

For dynamic memory synchronization, memory operators are connected through a synchronization token network in which nodes represent memory operators and edges represent synchronization tokens [12]. A token edge between two memory accesses indicates that the memory operators may access the same memory locations. Fig. 3(b) shows an example of a token network where dashed and solid lines represent token and data edges, respectively. A dashed line connecting $M[r2]$ to the multiplexer at the loop entry represents cross-iteration dependence with $M[r1+1]$ and $M2[r1+2]$ so that $M[r1+1]$ and $M[r1+2]$ can be executed after they receive a token from $M[r2]$. The token edge has the same handshaking protocol as the data flow.

When the speculation fails, the data-flow part of the pipeline does not need to be changed due to the handshaking approach. Only the token network must be reconfigured to execute memory accesses in the correct order. To this end, as shown in Fig. 3(c), we add two extra muxes for each token edge in order to enable or disable the edge. When the miss-speculation is detected, token edges among dependent memory accesses are disabled by deasserting sp bits. Consequently, when the speculation goes wrong, this approach makes it possible to roll back the speculated memory synchronization of the pipeline to correct memory synchronization by deasserting some cheap configuration bits, and the rest of the pipeline can remain intact.

IV. Related Works

Researchers have recently studied binary translation and synthesis for hardware acceleration. Stitt *et al.* [3] proposed a dynamic binary-level hardware/software partitioning for their proposed reconfigurable logic chip. Their tool includes a dynamic binary translator in which the microprocessor without interlocked pipelined stages binary is disassembled and then synthesized into the netlist. The binary translator transforms the binary dynamically, but they did not focus on dynamic optimization and memory access optimization. Critical Blue [10] and BINACHIP [14] provide commercial binary translation tools to create a custom processor from the executable binary. Mittal *et al.* [2] proposed an automatic binary translation from TI's DSP binary onto FPGAs.

Although there exists an extensive body of work on loop pipelining, we briefly outline the recent work targeting reconfigurable architectures. Weinhardt and Luk [15] present a pipeline vectorization method from C code without pointers to synthesize the hardware pipelines from C code. They also propose several loop transformations to exploit a large amount of parallelism and hardware specialization to increase circuit utilization. Budiu [12] presents a compiler framework CASH for the automatic synthesis of circuits from programs written in C. In the framework, he proposes new intermediate representation called Pegasus which unifies several features of other intermediate representations. His techniques have achieved the large amount of parallelism, exploiting the pipeline parallelism with memory access optimization techniques. However, his tool synthesizes the pipeline from C code with additional annotations to reveal memory dependence among pointers. Xilinx CHiMPS [13] provides a framework to synthesize the application-specific accelerator onto FPGAs with their CHiMPS Target Language (CTL). The CTL is an intermediate representation to abstract the architecture of the FPGAs for high-level language compilers. Each operator in the CTL is converted into an instance of a predefined hardware module. Data flow between the operators is automatically synchronized by FIFOs. Since the architecture of CASH and CHiMPS is based on the handshaking approach, the proposed speculative loop pipelining can be easily applicable to those architectures. There also exist some related works in compilers for modern CRAs such as ADRES [8], [16], XPP [11], Morphosys [17], and Montium Tile Processor [18].

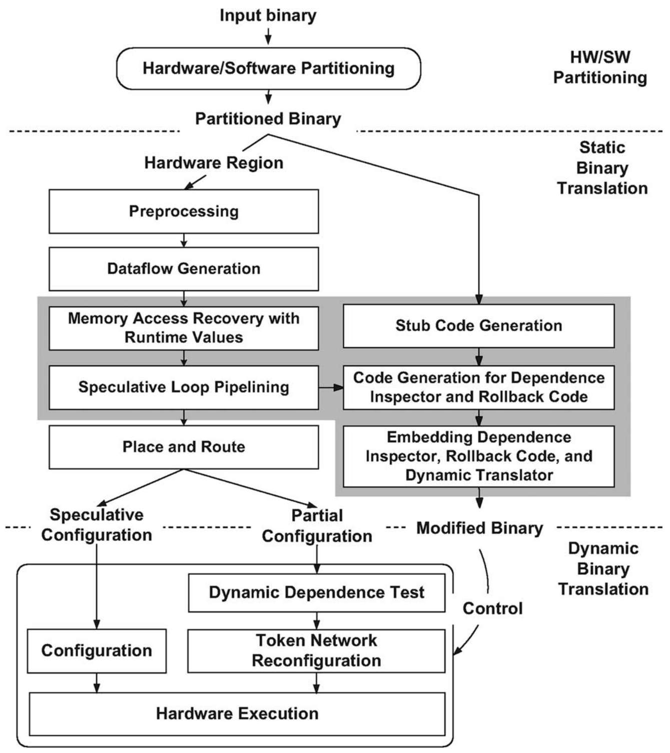

Input binary
Hardware/Software Partitioning
HW/SW Partitioning
Partitioned Binary
Static Binary Translation
Hardware Region
Preprocessing
Dataflow Generation
Memory Access Recovery with Runtime Values
Stub Code Generation
Speculative Loop Pipelining
Code Generation for Dependence Inspector and Rollback Code
Place and Route
Embedding Dependence Inspector, Rollback Code, and Dynamic Translator
Speculative Configuration
Partial Configuration
Modified Binary
Dynamic Binary Translation
Dynamic Dependence Test
Control
Configuration
Token Network Reconfiguration
Hardware Execution

Fig. 4. Overview of binary translation framework.

V. Binary Translation Framework

The design flow is divided into three parts: 1) HW/SW partitioning; 2) static binary translation; and 3) dynamic binary translation, as shown in Fig. 4. In the partitioning phase, critical loops are extracted for hardware implementation, and an instruction stream chosen for the hardware execution is referred to as a *hardware region*. Recently, several techniques have been studied for partitioning and loop extraction at the binary level [2], [3], and they can be easily incorporated into our framework.

A. Static Binary Translation Framework

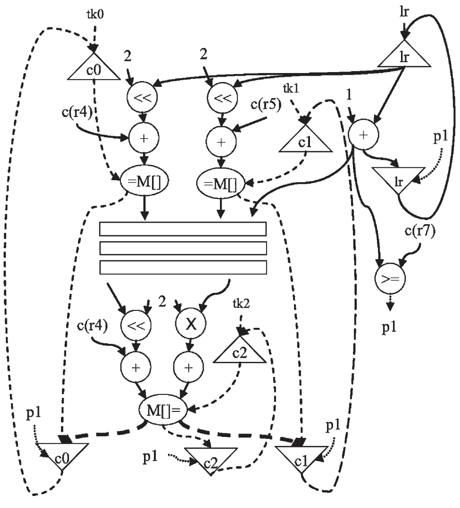
The static binary translator has two major roles: synthesizing the speculatively pipelined loops and generating the code fragments for the dynamic binary translator. The static translation consists of preprocessing, data-flow generation, speculative loop pipelining, and back-end to generate netlists and software/hardware interfaces.

1) Preprocessing: Preprocessing is the process of decoding instructions to a machine-independent form and recovering high-level semantics regarding control and data dependence. Decoding converts the instructions of the hardware region into a collection of register transfer lists (RTLs) with a hierarchical control flow graph. After the RTLs are generated, stack accesses are replaced with symbolic registers, because stack accesses increase the number of memory operations. We also apply traditional decompilation and loop detection techniques to recover the control flow, such as basic blocks and loops.

2) Data-Flow Generation and Memory Synchronization: We generate a data-flow graph from RTLs, using Pegasus, which is the well-defined data-flow intermediate representation [12] for the synthesis of the circuits. The input RTL is

```
L1:
r1 = M32[r6 + lr << 2];
r2 = M32[r5 + lr << 2];
r3 = r1 << 16;
r12 = r2 >>' 16;
r3 = r3 >> 16;
r0 = r12 * r3;
r2 = r2 << 16;
r1 = r1 >>' 16;
r2 = r2 >> 16;
r3 = r0 + r1 * r2;
r3 = r3 >>' 16;
r2 = r12 * r1 + r3;
M32[r4 + lr << 2];
lr = lr + 1;
lr < r7 ? jmp L1;
```

(a)



(b)

Fig. 5. Example binary and its generated data flow.

decomposed into hyperblocks [19]. Then, the control flow is transformed into a data flow. Fig. 5(a) and (b) shows an example RTL code and its generated data flow. Pegasus is an extension of static-single assignment (SSA) with memory dependence and predication. It borrows some operators from data-flow machines and gated SSA [20] to transform control flow into data flow. The ϕ function in SSA is replaced with one of the Mu(μ), Eta(η), and multiplexer operators. Mu operators are equivalent to ϕ functions at the entry of the hyperblock. Eta operators called gateways in the data-flow model are placed at the exits of the hyperblock to steer the produced data to other hyperblocks with predicates. Multiplexer operators are used to merge data with predicates inside hyperblocks. In Fig. 5(b), the example data flow is a single hyperblock for the loop. Mu and Eta are triangular and inverse triangular, respectively. Memory operations are connected to the token network for memory synchronization described in Section III-B. In Fig. 5(b), the dashed lines indicate the token flow.

3) Speculative Loop Pipelining: Compared to the existing loop pipelining techniques, our proposed speculative loop pipelining contains additional compilation phases, as shown in gray shade in Fig. 4. The memory access recovery is a step to recover memory access patterns from the binary to discover memory dependence. The speculative loop pipelining phase generates the optimistic netlist with the dependence inspectors and the rollback code for the miss-speculation. The stub code is a small code composed of a few instructions to transfer a control from the software execution to the dynamic binary translator. Sections VI and V-B give the details for the speculative loop pipelining and the dynamic binary translator, respectively.

B. Dynamic Binary Translation Framework

The dynamic translator controls the reconfigurable hardware during the hardware execution. Fig. 6(a) illustrates the flow of control in more detail. Right after the stub code transfers control from the software execution to the dynamic binary translator, it first saves a snapshot of the software context, such

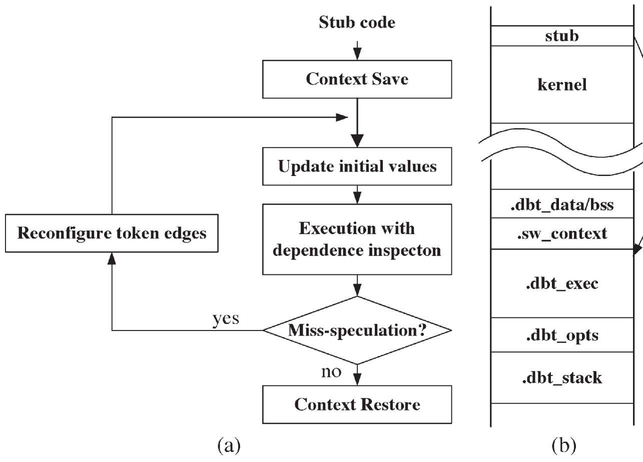


Fig. 6. Overview of dynamic translator. (a) How the dynamic translator works. (b) Memory layout and how the translator gains a control.

as machine registers and stack environment. Then, it swaps the stack pointer with a stack pointer pointing to its own stack environment in order to avoid interference with the software stack environment. Before the hardware execution, the dynamic translator updates the initial values of the data flow. Then, it executes the hardware with dependence inspectors. When the miss-speculation occurs, the token network is reconfigured for the correct memory synchronization. After the hardware finishes its computation, the dynamic translator restores the software context with values from the data flow.

The dynamic translator is provided as an object file, and the file is linked to the original binary located in an unused memory space, as shown in Fig. 6(b). The entry point into the dynamic translator is the routine *dbt_exec*. The sections *dbt_data* and *dbt_bss* contain the data of dynamic translator, and the section *sw_context* is occupied to save the software context. The section *dbt_opt* contains the dependence inspectors and rollback codes.

VI. Speculative Loop Pipelining

A. Memory Access Recovery and Summary

1) Deriving Loop Invariants: In many signal-processing applications, important values for memory analysis, such as the base address of memory accesses and loop counts, are loop invariants during the loop execution, although those values are not known at compile time. Deriving loop invariants is similar to constant propagation and runtime constant propagation in SSA [21]. The initial set in the propagation contains symbolic variables which represent values stored in software context. Then, it derives loop invariants, propagating the initial values through the data flow. The data flow in Fig. 7 shows a part of data flow generated from an image processing code. The values in the initial set are $c(r3)$, $c(r6)$, $c(r7)$, and $c(r8)$, and the derived loop invariants are shown along with the edges of the data flow.

It is not yet known what the derived loop invariants mean in the loop. For example, $c(r8) \times (c(7) - 2)$ and $c(r3) + c(r6) - 2$ correspond to the loop counts and the base address, respectively. However, the analysis results do not show if those

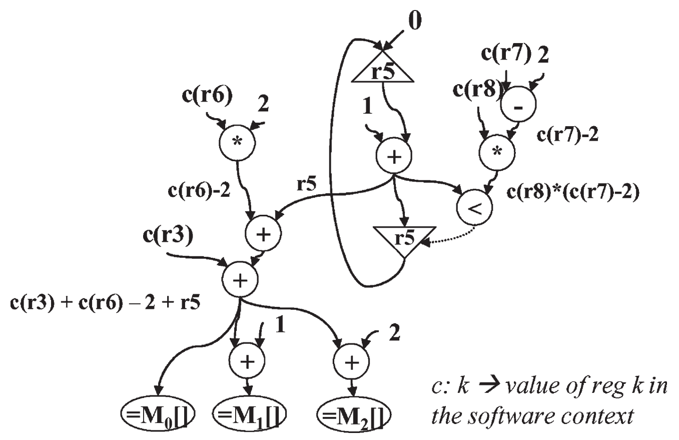


Fig. 7. Part of data flow generated from an image processing code in the TI image processing library. Loop invariants give hints to analyze memory access patterns.

Example code	Evolution of inp and i
do {	inp: $\diamond$ i: $\diamond$
if(f1()) {	
d2 = *(inp+2);	
d1 = *(inp+1);	
d0 = *inp++; }	
if(f2(d0,d1,d2))	inp: $\triangleright_{(4,0)}$
if($i \geq$ slen)	
break;	
i = i + 1;	
} while(i < len);	i: $\triangleright_{(1,1)}$
Stride of inp and i:	inp: $\triangleright_{(4,0)}$ i: $\triangleright_{(1,1)}$

Fig. 8. Example code and the changes of the data flow information in the monotonic evolution. The inp is a 32-bit pointer variable.

Algorithm 1 Algorithm for memory access recovery

```
1: procedure MEMORY_ACCESS_RECOVERY
2:     global_runtime_constant_propagation()
3:     for every entry node $n$ in the hardware region do
4:         evolution($n$,$null$)
5:     end for
6: end procedure
7: procedure EVOLUTION($node\ n$, $loop\ l$)
8:     $innerloop(l)$ : a set of outermost loops in $l$
9:     $header(l)$ : a set of mu nodes in the header of $l$
10:    $succ(n)$ : a set of successor node of $n$
11:    $pred(n)$ : a set of predecessor node of $n$
12:    $Stride(n)$ : a pair of maximum and minimum distance strides
13:    $Stride_{M/m}(n)$ : the maximum/minimum distance stride
14:    $lcnt(l)$: the loop counts of $l$ and $lcnt(l) \geq 0$
15:
16:    if $l$ is $null$ or ( $n \in header(l)$ and $evolve(l) = false$ ) then
17:        for all $l' \in innerloop(l)$ do
18:            $Stride(n) := \perp$
19:            for all $n \in header(l')$ do
20:                evolution($n$,$l'$)
21:            end for
22:            forward_substitution($l'$)
23:            bound_analysis($l'$)
24:            $lcnt(l')$ := estimate_loop_counts($l'$)
25:            estimate_range_of_variables($l'$, $lcnt(l')$)
26:            $evolve(l') := true$
27:        end for
28:    end if
29:    if $n$ is a join node such as mu and mux then
30:        $in_n := \vee out_{pred(n)}$
31:    end if
32:    if $n \in \{\ m \in header\ (\ l')\ \mid\ l' \in innerloop(l)\ \}$ then
33:        $out_n := in_n \circ (\ Stride_m(n) \times lcnt(l')\ )$
34:    else
35:        $out_n := f(in_n)$
36:    end if
37:    for all $s \in succ(n)$ do
38:        if $s$ is $header(l)$ then
39:            $Stride(n) := Stride(n) \vee out_n$
40:        else if $s$ is included $l$ then
41:            evolution($s$,$l$)
42:        end if
43:    end for
44: end procedure
```

loop invariants are related to the loop counts and the base address.

2) Memory Access Recovery: This section describes a recovery process which retrieves memory access patterns on the nested loops with loop invariants. It first keeps track of how the values of induction variables, such as loop indices and pointers in the data flow, change. DSP applications written in C/C++ often make extensive use of pointer arithmetic expressions. The pointers are usually induction variables. Such pointers increase or decrease monotonically during the loop execution. However, such pointer values do not change as strictly as loop indices do. For instance, in Fig. 8, it is difficult to know how many times inp is incremented during the loop execution. To cope with this problem, we devised a data-flow analysis based on monotonic evolution [22] in order to detect the monotonicity of variables and to compute the stride of those variables. The stride represents how the value of the variable changes for a single loop iteration. The monotonic evolution is incorporated in the function evolution with additional analysis in Algorithm 1.

The evolution function analyzes the nested loops from the innermost loop to the outer loops. First, it computes the monotonic evolution of the variable, which means how the value of each induction variable changes from the loop header. The data-flow information of the evolution can be one of the following values: $\perp$, $\top$, $\diamond$, $\triangleleft_p$, and $\triangleright_p$, which describe "no evolution," "unknown evolution," "the initial value for every variable," "monotonic decreasing," and "monotonic increasing," respectively. p represents the difference of the value between the loop header and the current operation, which is called the distance of the evolution. The evolution function computes two types of distance, the maximum distance and the minimum distance, because the distance may be a range due to control flow. Consequently, p is a tuple of (m,n), where m is the maximum distance and n is the minimum distance ($m > 0$, $n > 0$). The function f is a transfer function of the evolution information that interprets operations of the data flow and generates the evolution information. The operator $\vee$ joins the evolution information at multiplexer nodes and Mu nodes to compute the maximum distance and the minimum distance. For example, the example code in Fig. 8 shows a change of evolution information for the induction variable before interpreting the code in the current line and the stride of the variables. In the algorithm, the evolution function is invoked at line 20, and the analysis is performed at lines 29–43. Line 33 has an additional operator $\diamond$, which composites two evolutions, such that $\triangleright_{(m1,n1)} \diamond \triangleright_{(m2,n2)} = \triangleright_{(m1+m2,n1+n2)}$. In the outer loop, the inner loops are treated as an abstract node, and the value of the induction variable changes by the stride of the variable

V = $\{\perp, Z, \top\}$ where $n \in Z \geqq LB$

node	$\perp$	e_1	$\top$
ID	$\perp$	e_1	$\top$
LB w/ e_2	$\perp$	e_1	$\top$
UB w/ e_2	$\perp$	$min(e_1, e_2)$	e_2
UN	$\perp$	$\top$	$\top$

$\vee$	$\perp$	e_1	$\top$
$\perp$	$\perp$	e_1	$\top$
e_2	e_2	$max(e_1, e_2)$	$\top$
$\top$	$\top$	$\top$	$\top$

Fig. 9. Lattice arrangement, transfer function, and join operator for the bound analysis for increasing monotonic evolutions in order to find its upper bound. The vertex of lattice is defined to be a value whose range is from LB to ∞, which represents that the component is bounded by its lower bound, LB and its upper bound from LB to ∞. The lattice and tables for decreasing evolutions are constructed similarly.

in the inner loop. The composite operator is used to combine the evolution value right before the inner loop and the stride in the inner loop.

After the evolution function for the loop l is done, it is followed by a set of analysis in lines 22–26 so as to recover the loop counts and memory access patterns in the loop. First, the forward_substitution finds dependent induction variables which are linear equations of the found basic induction variables. The bound_analysis attempts to discover the bounds of induction variables, initial and final values. It is easy to find the initial value because many induction variables are usually assigned to their initial value outside the loop. On the other hand, it is not trivial to find the final value since the number of loop iterations is unknown. However, most loops have at least one induction variable whose final value is bounded by the inequality equation included in the loop exit branch. Accordingly, the bound analysis contains data-flow analysis, which gathers information from the Eta nodes, and their predicates are converted from branch instructions exiting the loop. Every monotonic evolution is a strongly connected component, and the data-flow analysis has to be solved for each component. Fig. 9 shows the values of data-flow information, the lattice representation, the transfer function, and the join operator. Nodes in the data flow are categorized as follows.

1) Identity node: No predicate is generated.
2) Upper bound node: A node has a descendant Eta node which has a predicate $p = !v < e$ to exit the loop, where v increases monotonically ($\Delta v > 0$).
3) Lower bound node: A node has a descendant Eta node which has a predicate $p = !v > e$ to exit the loop, where v decreases monotonically ($\Delta v < 0$).
4) Unknown bound node: A node has a descendant Eta node which has a complex predicate expression.

v is an induction variable on the component, and e is a loop invariant. The analysis for a component does not visit the loop header twice, and the bound information is joined at the loop header through back edges. The joined bound estimates the upper or lower bound of the component. This bound analysis is similar to the narrowing operator in abstract interpretation [23]. In most DSP applications, the boundary analysis results in one or few induction bounded variables such that $\min(I_v, F_v) < v < \max(I_v, F_v)$, where I_v and F_v are loop invariants for the initial and final values of variable v.

The loop counts are estimated to be $lcnt(l) = \lceil(\text{Range } of\ v/$ Minimum distance stride of $v)\rceil = \lceil((|I_v - F_v + 1|)/|Stride_m(v)|)\rceil$ from a variable whose bounds and strides are computed. The loop counts make it possible to estimate the range of other induction variables since the final value of the unbounded variable v' is defined to be $F_{v'} := I_{v'} + Stride(v)_M \times (lcnt(l) - 1)$.

At this point, the recovery process has recovered and estimated the loop counts and induction variables with loop invariants. An address expression for a memory access is converted into a sum-of-products form

$$\text{addr} = \Sigma_{m=1} f_m(i_1, i_2, \ldots, i_n) + C_R + f_{\text{cpx}} \tag{1}$$

where i_k represents an induction variable. f_m is a function defined by induction variables and loop invariants. C_R is the base address that is loop invariant. f_{cpx} is a complex expression, which is not a closed-form expression in terms of induction variables and loop invariants. If f_{cpx} is zero, the access pattern can be estimated at runtime, because the equation is defined by loop invariants. On the other hand, if a memory access has the complex expression, the access pattern cannot be analyzed, and it has may-dependence with all the other memory accesses. We partially deal with such irregular memory accesses by a hardware dependence test for common access patterns, as discussed in Section VI-D.

3) Access Region Descriptor: It is useful to summarize memory accesses in the loop nests for efficient memory dependence analysis. For the binary translation, descriptors for the summary have to provide a fast and lightweight dependence analysis on the linear memory space regardless of the data structures of program language. Although a number of summarizing techniques have been studied over the years, many of those are too expensive to solve the dependence analysis because of solvers such as linear program and constraint-based scheme. In addition, some of the related works are not suitable to analyze linearized memory accesses. Recently, a linear memory descriptor was proposed in [24]. This paper refines the linear memory access descriptor for the binary translation. The descriptor is formulated as

$$\text{ard} = \Delta + \tau = M_{k_{\text{bits}}}{}^{\delta_1}_{\sigma_1}{}^{\delta_2}_{\sigma_2} \cdots {}^{\delta_k}_{\sigma_k} \cdots {}^{\delta_n}_{\sigma_n} + \tau \tag{2}$$

where $M_{k_{bits}}$ represents the width of memory access, δ_k represents a tuple of the maximum and minimum distance strides at loop level k, and σ_k represents the maximum span of the access sequences at loop level k. A pair of δ_k and σ_k is called access dimension at loop level k where level 1 represents the outermost loop. τ represents an offset term. If the maximum and minimum distance strides are the same at every loop level, the access descriptor is called accurate because we can keep track of memory accesses accurately without complex control flow. For example, if variable inp in Fig. 8 has a type of 4-byte integer (32-bit pointer variable in high-level code), its access descriptor is computed to be as follows:

$$\mathrm{ard}(\mathrm{inp}) = M32^{(4,0)}_{c(\mathrm{len})\times 4} + c(\mathrm{inp}) \tag{3}$$

where $c(\mathrm{inp})$ and $c(\mathrm{len})$ are loop invariants representing the initial values of inp and len stored in the software context, respectively.

4) Memory Access Cluster: Data processing codes often have many memory accesses. For example, some kernel loops have tens of memory accesses per iteration. It is undesirable to discover all memory dependence dynamically between all possible pairs of memory references because the number of dependence inspection is too many to execute at runtime. Therefore, we classify memory access into several clusters called the memory access clusters. A memory access cluster is defined to be a set of memory accesses to a common memory segment. This is equal to a subset of an array and a dynamically allocated linear memory segment. Consequently, memory accesses in the same cluster are dependent, and memory accesses in different clusters are supposedly independent. The token network is designed for memory operations accessing the common segment to share Eta nodes and a single Mu node [12].

We currently classify memory accesses shifted by compile-time constants into a cluster. For example, the three read memory accesses $^*(\mathrm{inp}+2)$, $^*(\mathrm{inp}+1)$, and $^*\mathrm{inp}++$ in Fig. 8 are classified into a cluster because they have the same base address and compile-time constant offsets. Memory descriptors for the three accesses are as follows:

$$M32^{(4,0)}_{c(\mathrm{len})\times 4} + c(\mathrm{inp}) + C_{\mathrm{offset}} \tag{4}$$

where $C_{\mathrm{offset}} = \{0, 4, 8\}$. In other words, two memory accesses are in the same cluster, if the symbolic substraction of their access region descriptors is a compile-time constant.

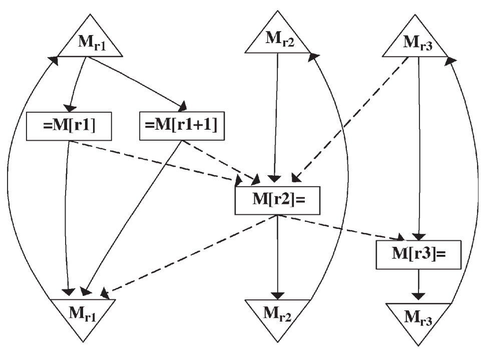
This approach is simple, yet practical. First, memory accesses in a cluster mostly access a common memory segment at a high level. If the addresses of memory accesses in the same cluster have non compile-time constant differences, the access range of the cluster can be large in the memory space. Second, we can make use of offset analysis to statically determine the dependence between intracluster memory accesses. Otherwise, it requires runtime decisions. Finally, the data-reuse technique, such as register promotion, computes the number of registers with the differences between the offset values so that the offset values should be compile-time constants in order to instantiate the fixed number of registers.

B. Speculative Token Network and Data Reuse for Pipeline Parallelism

1) Constructing the Token Network: The token network construction is divided into two major parts: constructing the token network for each memory access cluster and inserting intercluster token edges between the clusters. Fig. 10 shows an example assembly code and its token network. Memory accesses are classified into three clusters in the example code. Each cluster has a Mu node and an Eta node to carry a token.

```
L1:
  r5 = M[r1];
  r6 = M[r1+1]; r1++;
  r2 = f1(r5);
  r5 = M[r2];
  r7 = f2(r5,r6);
  M[r3] = r7; r3++;
  goto L1 if r4++ < 50;
```

(a)



(b)

Fig. 10. Example assembly code and its token network.

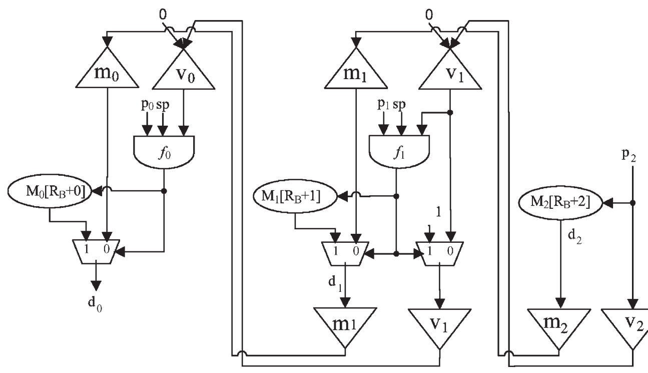


Fig. 11. Data reuse across loop iterations for example code in Fig. 7. $p0$, $p1$, and $p2$ are predicators for M_0, M_1, and M_2, respectively. f_0 and f_1 are Boolean functions such that $f_1(sp, p_1, v_1) = ((sp \cdot \overline{v_1})|\overline{sp})p_1$ and $f_0(sp, p_0, v_0) = ((sp \cdot \overline{v_0})|\overline{sp})p_0$.

Solid lines represent intracluster token edges. Since memory dependences in a cluster can be statically analyzed, token edges connecting memory operations in the same cluster do not have muxes on their handshaking signals, as shown in Fig. 3. After intracluster token networks are built, intercluster token edges represented by dashed lines are inserted. As mentioned in Section III-B, the intercluster token edges have muxes on their handshaking signals to support the speculative memory synchronization.

2) Data Reuse Across Loop Iterations: By accessing the same memory locations across several loop iterations, redundant memory operations in many DSP loops are replaced with registers by applying a register promotion technique [7]. Register promotion also assumes that the repeated memory accesses are not dependent to other memory accesses. This technique is also performed speculatively and can be easily rolled back. Fig. 11 shows that the load operations in Fig. 7 are promoted to contained registers in muxes. The transformed data flow runs in a speculative mode or a conservative mode controlled by bit sp. In the example, when the sp bit is set, the value of M_k is reused by M_{k-1} in the next iteration. Assuming that all the predicators for the memory accesses are true, M_1 is only executed in the first iteration, and M_0 is executed in the first two iterations because the initial values of v_0 and v_1 are zero. On the other hand, M_2 runs at every iteration. The sp bit is initially set to "1," and the dynamic translator resets it only if the dependence is discovered. If the sp bit is zero, M_0, M_1, and M_2 are performed at every iteration.

The register promotion is applicable for a set of memory operations that satisfies the following conditions (given with two M_i and M_j in the set).

1) M_i and M_j belong to the same level of loop.
2) δ_n is strictly monotonic (increasing or decreasing).
3) $\text{ard}(M_i)$ - $\text{ard}(M_j) = \delta_n \times k$ such that $k \in \mathbb{N}$.

$\text{ard}(M_i)$ represents the access region descriptor of M_i. We currently apply this technique only to the innermost loop.

C. Generating Dependence Inspectors

An inspector is a small code fragment or a hardware logic that discovers memory dependence dynamically between two memory access clusters. An inspector takes as input the values of loop invariants computed from the software context, compares the access patterns of two memory access clusters, and returns true if there is no dependence.

We classify dependence inspectors into software inspectors and hardware inspectors. A software inspector is a code fragment executed on a host processor, whereas a hardware inspector is a small logic attached to the computation pipeline. A software inspector runs with the loop in two ways: serial execution and concurrent execution. The serial execution executes a dependence inspector prior to the loop execution, which causes runtime overhead in time. The concurrent execution has zero overhead for the dependence inspection because it concurrently executes the loop and the inspector. However, it requires an extra local memory to temporarily store data. Then the stored data must be committed to the main memory after the loop execution. The concurrent execution makes it easy to design the dependence inspector since we do not need to worry about memory corruption due to speculative store operations. On the other hand, we have to prevent any speculative stores from being occurred in the serial execution. This paper covers the serial execution.

1) Stream Accesses With the Same Stride: The first inspector is based on a strong single index variable (SIV) test [25]. Strong SIV test generates an efficient runtime test when two access descriptors have the same stride. The test computes the distance between two memory accesses. Dependence exists only if the distance is shorter than the span of address during the execution of innermost loop. The details of inspector generation appear in Algorithm 2.

The input of the algorithm is a pair of access region descriptor sets for two clusters. The strides are the same and accurate. The dependence distance between two input descriptors is computed symbolically in line 2. A generated inspector first contains an inequality equation so as to find out if the access ranges of two clusters are overlapped during the loop execution in lines 5–10 extends the inspector with additional tests for two clusters whose access ranges are overlapped in the memory space. These tests can be applied when D_1 and D_2 have a single descriptor, respectively. The test in line 5 can prove independence when two memory accesses are interleaved. On the other hand, the test in lines 6–10 checks the direction of dependence distance when read and write accesses occur in a stream.

Algorithm 2 Inspector 1

```
1: procedure GENERATE_INSPECTOR_SIV($D_1$, $D_2$)
   $D_1 = \{d | d \text{ is } M_{k1}{}^{\delta_1}_{\sigma_1}{}^{\delta_2}_{\sigma_2} \ldots {}^{\delta_n}_{\sigma_n} + \tau_1 + C_l \text{ where } l \in 0 \ldots |D_1|\}$
   $D_2 = \{d | d \text{ is } M_{k2}{}^{\delta_1}_{\sigma_1}{}^{\delta_2}_{\sigma_2} \ldots {}^{\delta_n}_{\sigma_n} + \tau_2 + C_m \text{ where } m \in 0 \ldots |D_2|\}$
   descriptors in $D_1$ and $D_2$ are linear and accurate
   $rw\_path(d_1, d_2)$: $descriptor \times descriptor \rightarrow Boolean$ returns $true$ when
   the loop reads data with $d_1$, compute results with the data, and then store the results
   with $d_2$
2:     $dist \leftarrow \tau_1 - \tau_2 + C_l - C_m$
3:     $ins \leftarrow min(|dist|) > \sigma_n + max(k1, k2)/8 - 1$
4:     if $|D_1| = |D_2| = 1$ then
5:         $ins \leftarrow (ins) \vee (dist \neq 0 \wedge dist \neq \delta_n)$
6:         if $rw\_path(d_1, d_2)$ then
7:             $ins \leftarrow (ins) \vee (\neg(ins) \wedge ((\delta_n > 0 \wedge dist \geq 0) \vee (\delta_n < 0 \wedge dist \leq 0)))$
8:         else if $rw\_path(d_2, d_1)$ then
9:             $ins \leftarrow (ins) \vee (\neg(ins) \wedge ((\delta_n > 0 \wedge dist \leq 0) \vee (\delta_n < 0 \wedge dist \geq 0)))$
10:        end if
11:    end if
12:    generate_instructions($ins$)
13: end procedure
```

For example, the FIR code in Fig. 1 has two memory access clusters for input and output streams. Assume that the input stream has a single load operation in[0] and that the variables in, o, and len are loop invariants and mapped to the machine registers r_{in}, r_o, and r_{len} in the binary. Its access region descriptor is represented by $M^4_{32_{c(r_{\text{len}}) \times 4}} + c(r_{\text{in}})$, and an access region descriptor for the output stream is $M^4_{32_{c(r_{\text{len}}) \times 4}} + c(r_o)$, where $c(r_{\text{len}})$, $c(r_{\text{in}})$, and $c(r_o)$ represent the values of the registers stored in the software context. The generated inspector for these two clusters is as follows:

$$\begin{aligned}
\text{dist} &= c(r_{\text{in}}) - c(r_o) \\
\text{ins1} &= |\text{dist}| > c(r_{\text{len}}) \times 4 + 3 \\
\text{ins2} &= \text{ins1} \vee (\text{dist} \neq 0 \wedge \text{dist} \mod \delta_n \neq 0) \\
\text{ins} &= \text{ins2} \vee (\neg\text{ins2} \wedge \text{dist} \geq 0).
\end{aligned}$$

In many kernels, ins1 is true, and hence, ins and ins2 are not evaluated at runtime.

2) Monotonic Evolutionary Accesses: The inspector based on the SIV test can perform a dependence test only if input descriptors are accurate and their strides are the same at all the loop levels. Range test [26] is a dependence test which can handle monotonic access patterns. The range test generates symbolic inequality equations for comparing access ranges and proves independence symbolically with a permutation of loop nest. The range test in this paper is evaluated at runtime, and the process of test is simplified because of the restriction of execution time. Algorithm 3 shows the details of inspector generation based on the range test. In contrast with the previous inspector, the algorithm can handle two clusters with different strides. The procedure AccessRange returns a pair of symbolic expressions representing the lower and upper bounds of access range at the innermost level. An inspector is a compound of inequality equations to compare access ranges. If $\delta_k \times i_k$ exists in ins after symbolic simplification, access ranges are approximated in lines 7–9. The approximation exaggerates the access ranges in the innermost loop, same as the access ranges in the outer loops, so that false dependence can be detected when two clusters access a common stream with interleaved accesses at the outer loop.

Algorithm 3 Inspector 2

```
1: procedure GENERATE_INSPECTOR_RANGE($D_1$, $D_2$)
   $D_1 = \{d | d \text{ is } {M_{k1}}^{\delta_{11}}_{\sigma_{11}}{}^{\delta_{12}}_{\sigma_{12}} \ldots {}^{\delta_{1n}}_{\sigma_{1n}} + \tau_1 + C_l$ where $l \in 0 \ldots |D_1|\}$
   $D_2 = \{d | d \text{ is } {M_{k2}}^{\delta_{21}}_{\sigma_{21}}{}^{\delta_{22}}_{\sigma_{22}} \ldots {}^{\delta_{2n}}_{\sigma_{2n}} + \tau_2 + C_m$ where $m \in 0 \ldots |D_2|\}$
   descriptors in $D_1$ and $D_2$ are monotonic
2:     $r1$ = AccessRange($D_1$)
3:     $r2$ = AccessRange($D_2$)
4:     $ins = r1.UB < r2.LB \vee r2.UB < r1.LB$
5:     symbolic_simplify($ins$)
6:     if $\delta_k \times i_k$ terms exists in $ins$ then
7:         // $ins = lhs_1 < rhs_1 \vee lhs_2 < rhs_2$
8:         substitute $\delta_k \times i_k$ in $lhs$ with $max(\sigma_k, 0)$
9:         substitute $\delta_k \times i_k$ in $rhs$ with $min(\sigma_k, 0)$
10:    end if
11:    generate_instructions($ins$)
12: end procedure
13: procedure ACCESSRANGE($D$)
14:    $LB = \sum_{k=0}^{k=n-1} (\delta_k \times i_k) + min(\sigma_n, 0) + \tau + min(C_m)$
15:    $UB = \sum_{k=0}^{k=n-1} (\delta_k \times i_k) + max(\sigma_n, 0) + \tau + max(C_m)$
16:    return ($LB$, $UB$)
17: end procedure
```

For example, code idct in the TI's image library has a two-level nest loop. Assuming that instructions for the two-level nest loop are selected for hardware region, descriptors for two clusters and the generated inspector are as follows:

$$
\begin{aligned}
&{M_{16}}^{-128}_{-128\times c(r_n)}{}^{-16}_{-128} + c(r_i) + \{0, 2, \ldots, 12, 14\}\\
&{M_{16}}^{-128}_{-128\times c(r_n)}{}^{-2}_{-16} + c(r_o) + \{0, 16, \ldots, 96, 112\}\\
&-128 i_1 + c(r_i) + 14 < -128 i_1 - 16 + c(r_o) \vee\\
&-128 i_1 + c(r_o) + 112 < -128 \times i_1 - 128 + c(r_i)
\end{aligned}
$$

where r_n, r_i, and r_o are the loop count of the outer loop, a pointer variable pointing out the input stream, and a pointer variable pointing out the output stream, respectively. An iteration variable i_1 can be eliminated symbolically so that the access ranges are tested accurately at the innermost level. This inspector can also be extended with additional tests for some overlapping cases.

D. Hardware inspector

Software inspectors can test dependence between two memory access clusters if memory accesses in the clusters do not have any complex term of f_{cpx}. However, loops in DSP applications often contain complex address patterns such as indirect memory accesses, $M[M[r_1]]$. For such complex terms, we present a hardware inspector which is a small amount of logic concurrently executing with the computation pipeline. Traditional runtime dependence tests for irregular array accesses [27] have been studied in Fortran parallelizing compilers. However, they have different assumptions on the target architecture and input language.

The proposed hardware inspector is applicable for two memory accesses (or two memory access clusters). One is a stream or monotonic access, and the other one is an irregular access. This means that the hardware inspector compares the address values computed by the irregular memory access with the access region bound of the other memory access at every loop iteration. Fig. 12 shows two cases in which hardware inspectors are applied. Solid lines represent data edges, and dashed lines represent token edges regardless of the types of token edges. In the first case, M_1 has a complex address, and the hardware inspector checks if the address of M_1 is overlapped with the

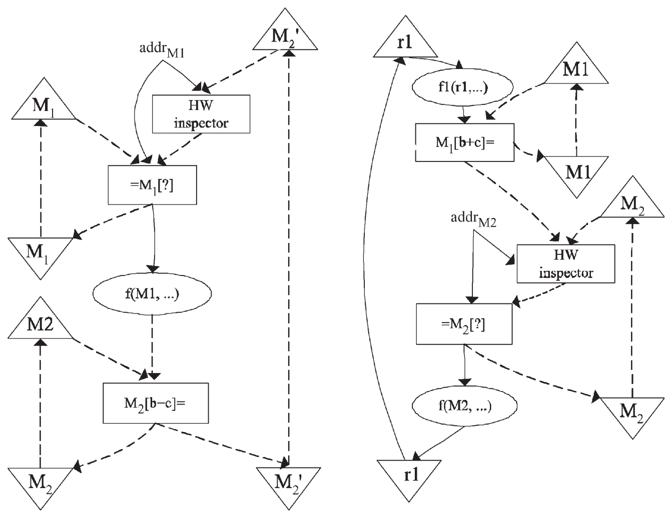


Fig. 12. Complex address at M_1 is observed by the hardware inspector.

Algorithm 4 The behavior of the hardware inspector

```
1: procedure HARDWARE_INSPECTOR_CLOCK_EVENT($addr$, $tk\_in$, $tk\_out$)
   $addr$ is the address of complex memory access
   $tk\_in$ is the input port of the inter-cluster token
   $tk\_out$ is the output port of token
   $LB$ and $UB$ are runtime constants representing the lower and upper bound of the other memory access
   $CNT$ is a counter serving like a semaphore
   insert($port$,$value$) inserts a packet with the value of $value$ into the $port$
   consume($port$) removes a packet in the $port$
2:     if $tk\_in.isReady$ then
3:         $CNT = CNT - 1$
4:     end if
5:     if $addr.isReady$ & $p.isReady$ & $!tk\_out.isStall$ then
6:         if $sp = false$ & $CNT = 0$ then
7:             $CNT = CNT + 1$
8:             insert($tk\_out$, 1)
9:             consume($addr$)
10:        else if sp = true then
11:            if $addr >= LB$ && $addr <= UB$ then
12:                $sp := false$
13:            else
14:                $CNT = CNT + 1$
15:                insert($tk\_out$, 1)
16:                consume($addr$)
17:            end if
18:        end if
19:    end if
20: end procedure
```

bounds of access region of M_2. The bounds of M_2 are runtime constants computed by the AccessRange function.

Algorithm 4 shows the behavior description at every clock event. The hardware inspector consists of a counter, two ALUs for the address comparison, and some combinational logic for the controller. LB and UB correspond to the access range of M_2, whereas addr corresponds to the runtime address value of M_1. The sp bit is set to "1" in the speculative mode, and every address value of the complex access is compared with the bounds of the other memory access. If the address is not overlapped, the output token is generated without synchronization, and hence, there is no dependence between the two accesses. Otherwise, the hardware inspector sets the sp bit to zero, or the hardware inspector sets the sp bit to zero. In this case, the counter serves as a semaphore to synchronize the memory accesses at the shared memory locations. The value of the counter represents how many store operations at M_2 must be performed prior to load operation at M_1. This is important to

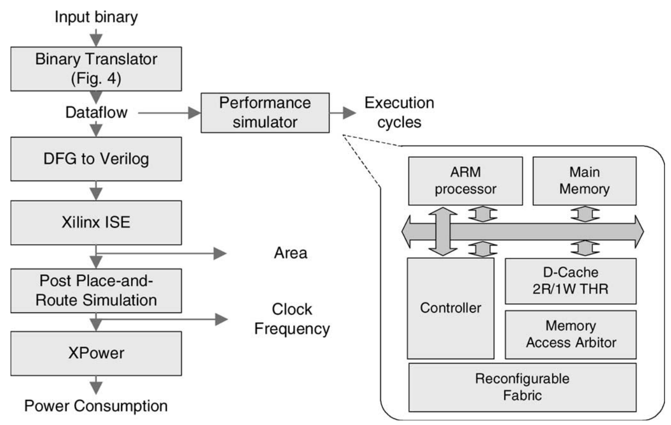


Fig. 13. Evaluation framework.

prevent M_1 from being starved for lack of tokens from M_2. Since there is no speculative store and the hardware inspector prevents the load operation from reading the wrong data, the data flow is side-effect-free.

The second case in Fig. 12(b) shows the data flow such that the data computed using the loaded data M_1 are consumed by store operation in the next iteration. Such a data flow is generated by scheduling optimizations such as software pipelining or by handcraft optimizations in high-level codes. The aforementioned technique can be applied similarly, but the compiler must guarantee that store operation must precede the load operation at the first iteration.

VII. Experiments

A. Experimental Setup

Fig. 13 illustrates our evaluation framework. It consists of two flows: performance simulation and mapping onto Xilinx FPGAs. We have developed a cycle-accurate simulator in SystemC to evaluate the execution cycle improvement on a reconfigurable SOC platform. The simulator contains three major modules: a cycle-accurate ARM module, a reconfigurable processor module, and a main memory. We extended the SimIT ARM simulator to communicate to other modules, and the ARM processor can control the reconfigurable processor module using ARM's coprocessor instructions.

The reconfigurable processor module is divided into a controller, a reconfigurable fabric, a data cache, and a memory access arbitrator. We assume that the reconfigurable fabric is composed of hardware-implemented modules of operations in the intermediate representation and that the modules are connected through the handshaking communication network. The latencies of the operators are similar as the latencies used for the performance simulator of CASH compiler framework [12]. Briefly, we assumed that ALU operations have a single-cycle latency and that control operations, such as Mu and Multiplexer, have a latency of $\log(n)$, where n is the number of inputs. When a memory operator has n token inputs, it takes $lon(n)$ cycles. We also assumed that there is no wire delay exceeding the clock period.

The controller provides a dynamic translator by transferring the initial values of the data flow into the reconfigurable fabric.

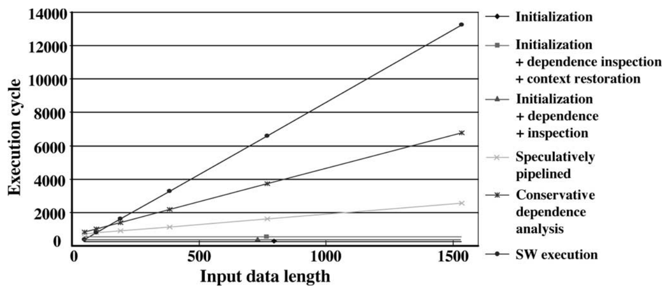


Fig. 14. Execution cycles for sobel filter.

After the computation is complete, the controller sets a flag for the dynamic translator to restore the software context.

In order to run any kernel which has multiple concurrent memory accesses with arbitrary addresses, we use an arbitration unit to share ports of the data cache. The arbitration unit takes a request and an address as input from a memory access node, schedules the request, and sends a done signal and the read data for a read request. We assume that the data cache has two read ports and one write port. When the data cache hit occurs, we assume that, in the reconfigurable fabric, the memory read operation takes two cycles and that the write operation takes one cycle. The line size of the cache is 32 bytes, and the cache-miss latency is 33 cycles. The main shared bus provides only a single memory port and can be used by other peripheral devices and other threads on the host processor.

The mapping flow takes as an input a Pegasus-based data flow and converts it into a verilog code. Since Pegasus is designed for ASIC synthesis, operations in the data flow are easily converted to the verilog code. We used Xilinx ISE 8.2 to map the verilog code onto FPGAs. After the place-and-route of the input design, we run timing simulation and XPower to report power consumption. To support multiple memory accesses, we implemented a quad-port data memory by using BRAM blocks. This can be extended into multiport data cache. Currently, Xilinx introduces a cache handling 16 read accesses with up to four memory accesses per cycle. In hardware implementation, the Mu and multiplexer operations have one-cycle latency regardless of the number of inputs instead of $\log(n)$.

B. Performance Simulation

We evaluated our techniques using kernel loops in multimedia benchmarks such as MediaBench, DSPStone, and TI's Image/DSP library. All benchmark programs were compiled with GNU C compiler cross-compiled to target ARM processors.

Fig. 14 shows the results of speculative loop pipelining for sobel image filter, including the number of execution cycles for hardware initialization, context restoration, and dependence inspection. The sobel filter has nine memory accesses clustered in six memory access clusters. One of those clusters consists of a single write access, and others only have read accesses. Therefore, five dependence inspectors are generated to check the dependences among those clusters. The dependence inspectors spend 114 cycles. Hardware initialization and the

TABLE I
EXECUTION CYCLE IMPROVEMENT AND SPEEDUP BY THE SPECULATIVE LOOP PIPELINING

Program	SW	$HW_{conservative}$				$HW_{speculative}$					SpeedUp		
	total	init	comp	final	total	init	dins	comp	final	total	$SP_{conservative}$	$SP_{speculative}$	SP_{dlp}
sobel_768	6603	270	3169	156	3595	270	114	1065	156	1605	1.83	4.11	2.23
fir4tap_256	5603	264	2665	157	3086	75	28	888	165	1356	1.81	4.13	2.28
idct8x8_64x7	17447	271	2655	1456	4382	271	33	1909	1457	3671	3.98	4.75	1.19
ycb_389	6045	273	1306	178	1757	286	99	785	179	1349	3.44	4.48	1.30
fir_dsp_256	2038	252	1394	153	1811	258	0	1394	153	1805	1.125	1.129	1.00
wvec_256	11901	264	3454	159	3877	274	51	1562	161	2048	3.06	5.81	1.89
mul32_512	5962	264	2665	157	3086	273	40	1052	160	1525	1.93	3.90	2.02
adpcmd_1000	47842	264	25315	152	25731	342	26	9719	157	10185	1.85	4.69	2.53
edge_64x64	155652	265	64640	385	65104	377	101	27264	182	27924	2.38	5.57	2.34

context restoration run for 270 and 156 cycles, respectively. However, the overheads are amortized over the pipeline cycles. The speculative loop pipelining speeds up the filter 2.23 times compared to the conservatively pipelined method and 4.11 times compared to the software execution.

Table I shows the performance results of several benchmark programs. The columns represent execution cycles for software execution (SW) on the ARM simulator, hardware execution with the conservative memory analysis ($\mathrm{HW}_{\mathrm{conservative}}$), hardware execution with the speculative loop pipelining ($\mathrm{HW}_{\mathrm{speculative}}$), and speedup over the software execution and the conservatively pipelined data flow (SpeedUp). In hardware execution, the terms init, dins, comp, and fin show the number of execution cycles to transfer runtime constants for the hardware initialization, the number of cycles to run dependence inspectors, the number of cycles to run the pipeline, and the number of cycles to update the software context on the ARM processor, respectively. In the speedup columns, $\mathrm{SP}_{\mathrm{conservative}}$ and $\mathrm{SP}_{\mathrm{speculative}}$ show the speedup of $\mathrm{HW}_{\mathrm{conservative}}$ and $\mathrm{HW}_{\mathrm{speculative}}$ over the software execution, respectively. $\mathrm{SP}_{\mathrm{dlp}}$ shows the speedup of $\mathrm{HW}_{\mathrm{conservative}}$ to $\mathrm{HW}_{\mathrm{speculative}}$. In addition, the numbers following the program name represent the length of input data in words for a single execution of the kernel loop. In practice, the loop with the conservative analysis results in the same code quality as the unpipelined loop due to memory dependences between load and store operations.

The results in column $\mathrm{SP}_{\mathrm{dlp}}$ show the effectiveness of the speculative loop pipelining technique. The speedups range from 1.19 for idct to 2.53 for adpcmd, except for fir. The speedup of fir_dsp is 1.0 because the dynamic translator detected that two of the three memory access clusters had loop-carried dependence. The overhead for hardware initialization and context restoration is usually amortized by the pipeline cycles. However, idct has a significant amount of cycles to restore the values. This is because idct has tens of stack locations, which require 1457 cycles to transfer the values of those stack locations to the software context. This overhead can be eliminated by live-out variable analysis, which remains as one of our future work. idct source code has only few variables used by the code following the loop, and hence, it requires only tens of cycles to restore the values.

Software dependence inspectors cause overheads up to 101 cycles. These overheads are recovered by the pipelined execution. Also, the pipeline for adpcmd has a hardware dependence inspector since the binary has an irregular memory access to access a lookup table.

TABLE II
DECREASE IN THE NUMBER OF MEMORY ACCESSES BY DATA REUSE

	w/o reuse	w/ reuse	reduction
edge	35072	15552	55%
sobel	1982	1525	23%
fir4tap	1535	515	66%

The data-reuse technique shows the decrease in the number of dynamic memory accesses, as shown in Table II. Our technique reduces the number of memory accesses by 55% for edge. edge has nine memory accesses in its loop body, which results in I/O bound. When the data reuse is applied, the number of memory accesses is decreased to four.

The current implementation of the binary translator lacks traditional optimization techniques, so that the speedups in $\mathrm{SP}_{\mathrm{speculative}}$ and $\mathrm{SP}_{\mathrm{dlp}}$ can be improved by additional optimization techniques. In particular, pipeline balancing techniques [28] are an important optimization to avoid pipeline stalls due to mismatches between the data production and consumption rates. Such pipeline stalls are another major problem in pipelined circuits. The pipeline stalls can be reduced by inserting buffers between pipeline stages so that the data producer can proceed without waiting for the consumer to be completed. In our binary translator, we have implemented a simple heuristic based on [29]. This technique deals with the pipeline stalls only for the datapath of the loop index variable. Pipeline stalls in the rest of the data path are not optimized. Consequently, the pipelines generated by our binary translator have longer loop initiation interval, compared to highly balanced pipelines. In addition, the speedups depend on the number of I/O resources. The number of memory ports in the simulator is two memory ports and one write port. Hence, some pipelines are I/O bounded. For example, idct8x6 has 16 pairs of read and write accesses. We evaluated the performance, increasing the number of the read–write pairs. When the number of pairs of the memory ports is four, the number of execution cycles is improved by 30%. Since the pipeline is not highly optimized, the performance is not I/O bounded for four pairs of read and write ports.

C. Hardware Implementation

We have mapped two benchmark programs fir and sobel onto Xilinx Vertex-4, xc4vlx25 to report the area, clock frequency, and power consumption. In hardware implementation, the fir has three taps, and the input sizes of fir and sobel are 64 and 384 bytes, respectively. Since the implemented binary translator lacks traditional compiler optimizations, we applied

TABLE III
EXECUTIME CYCLE, AREA, CLOCK FREQUENCY, AND POWER CONSUMPTION FOR THE SYNTHESIZED LOOPS

Benchmark	Pipeline strategy	Executime time		Maximum frequency		Area	Power/Energy @ 100MHz	
		II	$Cycles$	$f_{datapath}$	f_{mem}	$Slices$	$Power$	$Energy$
$sobel$	Speculative	4	527	162MHz	324MHz	944(8%)	81mW	4.269e-7J
	Conservative	12	1529	159MHz	318Mhz	881(8%)	46mW	7.033e-7J
fir	Speculative	4	256	193Mhz	386Mhz	488(4%)	51mW	1.305e-7J
	Conservative	9	576	204Mhz	408Mhz	359(3%)	33mW	1.900e-7J

few optimization techniques such as live-out variable analysis and pipeline balancing before generating the verilog code. Table III shows the results of the hardware implementation. In the second column, Speculative represents that the loop is speculatively pipelined and Conservative represents that the loop is pipelined with the conservative memory analysis. For speculatively pipelined loops, we used hardware inspectors instead of software inspectors in order to measure the overhead of the hardware inspector. II indicates the initiation interval of the loop. f_{datapath} and f_{mem} represent the maximum frequency of the data path and the quad-port memory, respectively. These numbers are reported by the timing analysis of Xilinx ISE. However, the timing simulation on the generated verilog code reports more accurate minimum clock periods (increase by more than 1 ns). Correspondingly, the maximum frequencies decrease. The memory module requires the doubled clock frequency because the quad ports are achieved by time-multiplexing a dual-port BRAM.

The speculatively pipelined loops have much less execution cycles due to the less initiation intervals. For fir, when the fir has additional taps, the initiation interval of the speculatively pipelined loop is still four cycles because the memory bandwidth is the same due to the data-reuse technique. On the other hand, the initiation interval of the pipelined loop with the conservative analysis increases because the data-reuse technique cannot be applied and additional pipeline stalls occur as many as the increasing number of pipeline stages between the load operations and the store operation. The hardware inspector causes time and area overhead. Since the hardware inspector is added to the token cycles, which are usually critical paths, it increases the length of the critical paths by one cycle, an additional cycle for the initiation interval. The pipelines with the speculation approach occupy more slices (7%–35%) due to the hardware inspector. For fir, when the fir has additional taps, the area overhead can be significantly reduced because it does not require more hardware inspectors for additional taps. The loops with the conservative analysis consume less power because their initiation intervals are much longer than the initiation intervals of the speculatively pipelined loops, and signals do not change as frequently as the speculatively pipelined loops. However, energy consumption for the speculatively pipelined loop is less than the energy consumption of the pipelined loop with the conservative analysis.

In this paper, resource sharing is not applied. However, since the loop initiation usually takes a few cycles due to strongly connected components with loop-carried dependences and since operations do not take their input at every cycle, sharing hardware resources may reduce the area. In particular, when the developer wants to synthesize the loop with the conservative memory analysis, resource sharing may significantly save the area due to the long initiation interval. Incorporating resource sharing and dynamic memory synchronization remains as a future work.

VIII. CONCLUSION

This paper presents a framework of binary translation for hardware acceleration and proposes the speculative loop pipelining with the runtime dependence tests to overcome the conservative memory analysis of the static binary translation. We presented promising results on signal processing kernels, showing a maximum speedup of 2.53 in the performance simulation. In addition, we have evaluated our technique at hardware level implementation on FPGA devices and achieved comparable clock frequency and power consumption compared to the conservative method while achieving significant improvement in the throughput. Further work will combine the speculative loop pipelining presented in this paper with the loop transformation techniques which can also involve outer loops into the loop pipelining such as loop tiling, loop merging, and loop distribution.

REFERENCES

[1] *Tutorial: Dynamic Binary Translation and Optimization*. [Online]. Available: http://www.microarch.org/micro33/tutorial/tutorial.html

[2] G. Mittal, D. C. Zaretsky, X. Tang, and P. Banerjee, "Automatic translation of software binaries onto FPGAs," in *Proc. DAC*, San Diego, CA, 2004, pp. 389–394.

[3] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: A first approach," in *Proc. DAC*, 2003, pp. 250–255.

[4] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm, "A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture," in *Proc. CASES*, Atlanta, GA, 2001, pp. 116–125.

[5] D. Koes, M. Budiu, G. Venkataramani, and S. C. Goldstein, "Programmer specified pointer independence," in *Proc. Workshop Memory Syst. Performance*, Washington, DC, Jun. 2004, pp. 51–59.

[6] S. Oh and T. G. Kim, "Memory access optimization of dynamic binary translation for reconfigurable architectures," in *Proc. ICCAD*, San Jose, CA, Nov. 2005, pp. 1014–1020.

[7] M. Budiu and S. C. Goldstein, "Inter-iteration scalar replacement in the presence of conditional control-flow," in *Proc. Workshop Optimizations DSP Embed. Syst.*, San Jose, CA, Feb. 2005, pp. 20–29.

[8] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study," in *Proc. DATE*, Feb. 2004, pp. 1224–1229.

[9] M. Mishra, T. J. Callahan, T. Chelceam, G. Venkataramani, M. Budiu, and S. Copen, "Tartan: Evaluating spatial computation for whole program execution," in *Proc. ASPLOS*, San Jose, CA, Oct. 2006, pp. 163–174.

[10] Criticalblue. [Online]. Available: http://www.criticalblue.com

[11] PACT, *Xpp-iii Processor Overview*, 2006. [Online]. Available: http://www.pactxpp.com

[12] M. Budiu, "Spatial computation," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, Dec. 2003. Tech. Rep. CMU-CS-03-217.

[13] D. Bennett, E. Dellinger, J. Mason, and P. Sudarajan,"An FPGA-oriented target language for hll compilation," in *Proc. Reconfigurable Syst. Summer Inst.*, Urbana, IL, Jul. 2006. [Online]. Available: http://www.ncsa.uiuc.edu/Conferences/RSSI/agenda.html

[14] *Binachip*. [Online]. Available: http://www.binachip.com

[15] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Trans. Comput.-Aided Design*, vol. 20, no. 2, pp. 234–248, Feb. 2001.

[16] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proc. DATE*, Paris, France, Mar. 2003, pp. 296–301.

[17] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi, "Design and implementation of the morphosys reconfigurable computing processor," *J. VLSI Signal Process. Syst.*, vol. 24, no. 2/3, pp. 147–164, Mar. 2000.

[18] *Montum Tile Processor*. [Online]. Available: http://www.recoresystems.com

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. MICRO*, Portland, OR, 1992, pp. 45–54.

[20] K. J. Ottenstein, R. A. Ballance, and A. B. Maccabe, "The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proc. PLDI*, White Plains, NY, 1990, pp. 257–271.

[21] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, "Fast, effective dynamic compilation," in *Proc. PLDI*, 1996, pp. 149–159.

[22] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua, "Monotonic evolution: An alternative to induction variable substitution for dependence analysis," in *Proc. Supercomputing*, Sorrento, Italy, 2001, pp. 78–91.

[23] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. Conf. Record POPL*, Jan. 1977, pp. 238–252.

[24] Y. Paek, J. Hoefliner, and D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 1, pp. 65–109, Jan. 2002.

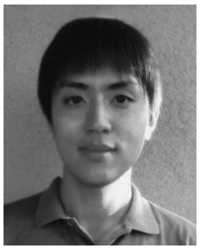
[25] R. Allen and K. Kennedy, *Optimizing Compiler for Modern Architectures: A Dependence-based Approach*. San Mateo, CA: Morgan Kaufmann, 2001.

[26] W. J. Blume, "Symbolic analysis techniques for effective automatic parallelization," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois at Urbana-Champaign, Urbana, IL, 1995.

[27] L. Rauchwerger, "Run-time parallelization: A framework for parallel computation," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois at Urbana-Champaign, Urbana, IL, 1995.

[28] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proc. ICCAD*, San Jose, CA, Nov. 2006, pp. 724–729.

[29] J. M. P. Caroso, "Self loop pipelining and reconfigurable dataflow arrays," in *Proc. SAMOS*, Samos, Greece, Jul. 2004, pp. 234–243.



Sejong Oh received the B.S. and M.S. degrees in electrical engineering and computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2000 and 2002, respectively. He is currently working toward the Ph.D. degree in the Department of Electrical Engineering and Computer Science, KAIST.

His research interests include optimizing compiler, design automation for embedded systems, and reconfigurable computing.

Tag Gon Kim (M'83–SM'93) received the Ph.D. degree in computer engineering with specialization in systems modeling and simulation from the University of Arizona, Tucson, in 1988.

He was a Full-time Instructor with the Communication Engineering Department, Bookyung National University, Pusan, Korea, between 1980 and 1983, and an Assistant Professor with the Electrical and Computer Engineering, University of Kansas, Lawrence, from 1989 to 1991. He joined the Electrical Engineering Department, Korea Advanced Institute of Science and Technology, Daejeon, Korea, in fall of 1991, as an Assistant Professor, where has been a Full Professor with the Department of Electrical Engineering and Computer Science since fall of 1998. He has published more than 150 papers on systems modeling, simulation, and analysis in international journals/conference proceedings. He is the Coauthor (with B.P. Zeigler and H. Praehofer) of the book *Theory of Modeling and Simulation (2nd ed.)* (Academic Press, 2000). His research interests include methodological aspects of systems modeling and simulation, analysis of computer/communication networks, and development of simulation environments.

Dr. Kim is a Senior Member of SCS and a member of Eta Kappa Nu. He was the Editor-in-Chief of SIMULATION: Transactions of SCS published by Society for Computer Simulation International (SCS).

Jeonghun Cho (S'96–M'03) received the B.S. degree in electrical engineering, the M.S. degree, and the Ph.D. degree in electrical engineering and computer science (EECS) from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1996, 1998, and 2003, respectively.

He was with the MCU Application Team of Hynix Semiconductors, Korea, where he worked on C Compiler development for 8-bit microprocessors. He is currently an Assistant Professor with the School of EECS, Kyungpook National University, Taegu, Korea. His research interest includes optimized compiler, operating system, and design automation for embedded systems and reconfigurable computing.

Dr. Cho is a member of ACM and IEEE.

Elaheh Bozorgzadeh (S'01–M'03) received the B.S. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 1998, the M.S. degree in computer engineering from Northwestern University, Evanston, IL, in 2000, and the Ph.D. degree in computer science from the University of California at Los Angeles, in 2003.

She is currently an Assistant Professor with the Department of Computer Science, University of California, Irvine. She has coauthored more than 35 conference and journal papers. Her research interest includes VLSI/FPGA CAD, design automation for embedded systems, and reconfigurable computing.

Dr. Bozorgzadeh is a member of ACM and IEEE. She received the Best Paper Award from the IEEE FPL in 2006.