Memory Access Optimization of Dynamic Binary Translation for Reconfigurable Architectures

Se Jong Oh, Tag Gon Kim
Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology (KAIST), Korea
sjoh@smslab.kaist.ac.kr, tkim@ee.kaist.ac.kr

Abstract

Recently, reconfigurable architectures, which outperform DSP processors, have become important. Although many compilers have been developed on a source-level, there are several practical benefits to translating the binary targeted to popular processors onto reconfigurable architectures. However, the translated code could be less optimized. In particular, it is difficult to optimize memory accesses on a binary to exploit pipeline parallelism. This paper introduces dynamic binary translation and memory access optimization to overcome the limitations of static binary translation for reconfigurable architecture. The experimental results show a promising speedup up to 3.02 compared with the code whose memory accesses is not optimized in the pipeline fashion.

Keywords: reconfigurable architecture, binary translation, dynamic optimization, memory access optimization

1. Introduction

Recently, binary translation is becoming popular in general-purpose architecture. In binary translation, the developer produces a standard binary without considering the underlying hardware, and then the virtual machine or binary translator on the hardware runs or translates the binary. Binary translation enables platform-independent design in which the application development is easy and fast, and single binary runs on heterogeneous hardware over the network.

In reconfigurable architectures, binary translation is not yet popular. Conventional design tools work on high-level source code or require the developer to write the netlist for high performance. However, binary translation is attractive for both practical and commercial reasons. First, reconfigurable architectures have different respective design flows. A company's existing tools may not be easily incorporated with a new design flow [3], and developers must migrate their applications, consuming the time and effort. Second, some compilers use dialects of standard language or their own annotations [2][1]. In addition, many critical kernels are often written in assembly. Furthermore, assuming future scenarios like network reconfiguration [5], the developer does not have the detail specification of underlying hardware, and the amount of available resource in reconfigurable hardware can vary dynamically for multi-threaded applications [6]. It is therefore desirable to translate the binary onto reconfigurable architectures. One of drawbacks in binary translation is that the code could be less optimized due to a lack of high-level information. In particular, memory access optimization for hardware synthesis is the key to achieve pipeline parallelism, but the loss of high-level information usually results in inefficient pipelines.

To overcome the limitation of memory access optimization in the binary translation, we introduce a dynamic binary translation exploiting pipeline parallelism for the platform composed of an ARM processor and a reconfigurable architecture. To achieve fast dynamic optimization, our binary translator consists of a static binary translator and a dynamic binary translator. The static translator generates most of the netlists for the execution and a small amount of partial netlists which can be configured at runtime according to the result of dynamic memory access analysis. The dynamic translator performs the analysis at runtime and configures the hardware partially with the partial netlists.

2. Related Works

Some recent researchers have already studied memory access optimization and binary translation for reconfigurable architectures. Yajun Ha et al [5] focused on a platform-independent development framework for reconfigurable systems. In their framework, the application developer designs a functional model, and partitions it into a software part and a hardware part. The software part is compiled into a software bytecode for a

software virtual machine. The hardware part is transformed into hardware bytecode on the abstract FPGA model. They introduced the concept of binary translation in the client, but they did not describe the process of compilation.

Greg Sitt et al [3][4] proposed dynamic binary-level hardware/software partitioning on the platform composed of a MIPS processor and their own reconfigurable logic chip. Their tool includes a dynamic binary translator in which the MIPS binary is disassembled and then synthesized into the netlist. Although they achieved a promising speedup, their reconfigurable hardware is simpler than commercial platforms so that they can synthesize loops which have a body implemented in a single cycle. In addition, their binary translator dynamically transforms the binary, but they did not focus on dynamic optimization and memory access optimization.

Mihai Budiu et al [9] presented memory access optimization techniques for hardware synthesis. Their techniques achieved a high degree of parallelism, exploiting the pipeline parallelism. However, their tool synthesizes the pipeline from C code with additional annotations to reveal memory dependences among pointers.

The contributions of this paper are as follows. We present the dynamic binary translator which exploits the pipeline parallelism on the binary. Our memory access optimization technique makes use of runtime information to overcome the limitation of static memory analysis on the binary; in addition, our techniques can be used in high-level language compilers instead of sophisticated static pointer analysis.

3. Binary Translation Framework

3.1. Overview

First, we describe the overview of our binary translation framework as shown in Figure 1. The design flow is divided into the side of the application developer and the side of the reconfigurable system similar to the network reconfiguration platform [5]. In the side of the application developer, the developer produces an executable binary and the result of the partitioning. The partitioning phase partitions the program roughly because the specification of client is unknown. The result of partitioning has the list of program counters pointing to program parts which the developer wants to execute on the hardware. We call an instruction stream chosen for the hardware execution a *hardware region*.

The side of a reconfigurable system contains a static binary translator and a dynamic binary translator. We will call them *static translator* and *dynamic translator*,

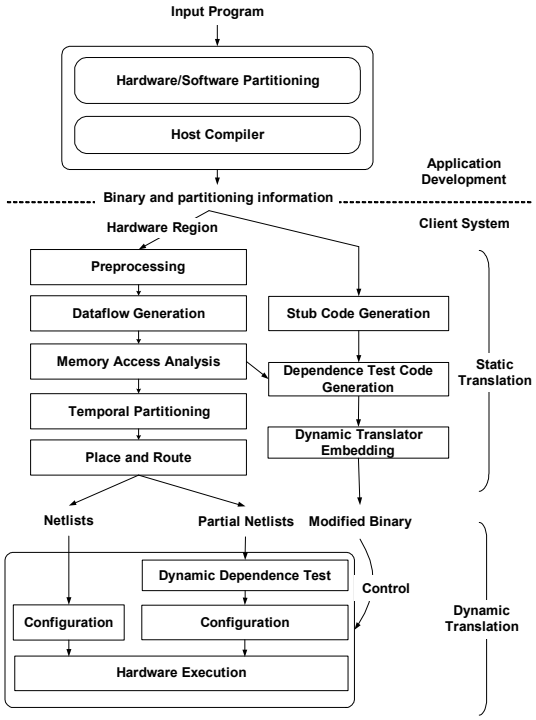


Figure 1 Overview of proposed binary translation

respectively, for the rest of paper. The static translator is executed once right after the binary is downloaded to the reconfigurable system. The static translator has two major roles: netlist generation and code generation for the dynamic translator. The netlist generation phase performs several steps from preprocessing to place and route. During the netlist generation steps, memory access analysis is a step to recover and analyze memory access patterns to discover memory dependences. Removing redundant memory dependences is essential to increase the amount of parallelism in the synthesized pipeline. To discover memory access patterns accurately, the analysis utilizes runtime information. The memory analysis statistically analyzes memory dependences as much as possible and then generates a small amount of code fragments to analyze memory dependences at runtime. These code fragments are called dependence test code.

The result of netlist generation contains two netlists: a template netlist and partial netlist. All configurations in the template netlist must be configured for the execution, while the partial netlist depends on the result of dependence test codes. The amount of the partial netlist is small for fast partial reconfiguration.

The code generation generates native instructions for stub code and dependence test code. The stub code is a small code fragment which transfers a control from the software execution to the dynamic translator. All code fragments and the dynamic translator are embedded into the binary.

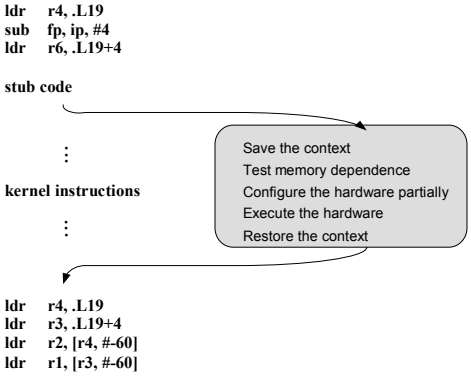


Figure 2 The flow of control

Figure 2 illustrates the flow of control during the program execution, assuming that the template netlist is configured at the first execution time. Right after the dynamic translator is called, it first saves a snapshot of the program context, such as machine registers and stack environment. It then calls dependence test codes to discover memory dependencies and configure the hardware with the partial netlist if needed. After the hardware execution finishes, values in the saved context are modified according to the result of hardware execution, and then the context is restored.

3.2. Preprocessing

Preprocessing is the process of decoding instructions to a machine-independent form and some code transforms. Decoding converts the instructions of hardware region into a collection of machine-independent register transfer lists (RTLs) with a hierarchical control flow graph (HCFG). Machine-independent RTLs and the HCFG make other phases of preprocessing retargetable, and they are converted to a dataflow graph in the next step. We applied traditional decompilation and loop detection techniques to recover the control flow such as basic blocks and loops. Figures 3-(a)(b)(c) show a data processing code from TI's DSP library, native instructions of the loop compiled by a GNU C compiler, and machine-independent RTLs transformed from the instructions, respectively.

After machine-independent RTLs are generated, optimizations phases transform them to increase the amount of parallelism. One of them is to eliminate stack accesses. RTLs have stack access to load and store local variables in high-level languages. These stack access can be replaced with symbolic registers. It is important to eliminate stack accesses because stack accesses increase the number of memory operations. These symbolic registers are initialized with values in a stack by the dynamic translator.

3.3. Dataflow Generation and Memory Synchronization

We generate a dataflow graph from RTLs, using Pegasus, which is the well-defined dataflow intermediate representation [11]. Pegasus is self-contained, enabling a complete synthesis of the circuits, without requiring further information. Figure 3-(d) gives a dataflow graph generated from the RTLs.

Pegasus provides control-flow operations, mu, eta, and a multiplexer to transform control-flow on RTLs into data-flow. In Figure 3-(d), mu and eta are triangular shaped and inverse-triangular shaped, respectively. Multiplexers are not necessary in the example code. Mu and multiplexer correspond to Static-Single Assignment ϕ functions, and eta is introduced in Gated-Single Assignment [8].

Memory operations need additional dependence edges because they have side-effects through memory. Such edges represent may-dependence between memory operations. Memory dependences are represented by

```
void DSP_mult32
(int*x, int* y, int* r, int nx) {
{
for (int i = 0; i < nx; i++) {
  short          a_hi, b_hi;
  unsigned short  a_lo, b_lo;
  int            hihi, lohi, hilo, hllh;

  a_hi = (short)(x[i] >> 16);
  b_hi = (short)(y[i] >> 16);
  a_lo = (unsigned short)x[i];
  b_lo = (unsigned short)y[i];

  hilo = a_hi * b_lo;
  lohi = a_lo * b_hi;
  hllh = (hilo + (long)lohi)>>16;

  r[i] = hihi + hllh;
}}
```

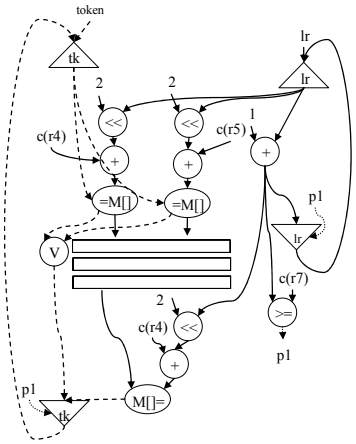
(a) Example code

```
ldr   r1, [r6, lr, lsl #2]
ldr   r2, [r5, lr, lsl #2]
mov   r3, r1, lsl #16
mov   ip, r2, asr #16
mov   r3, r3, lsr #16
mul   r0, ip, r3
mov   r2, r2, lsl #16
mov   r1, r1, asr #16
mov   r2, r2, lsr #16
mla   r3, r2, r1, r0
mov   r3, r3, asr #16
mla   r2, ip, r1, r3
str   r2, [r4, lr, lsl #2]
add   lr, lr, #1
cmp   lr, r7
blt   .L5
```

(b) native instructions

```
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;
```

(c) RTLs



(d) Generated dataflow

Figure 3 Example binary and dataflow

synchronization tokens in Pegasus. A token is a single value information. Memory operations wait for the token before executing. In Figure 3-(d), dashed lines indicate token flow, and the operator V combines tokens. Load and store operations in the figure have dependence across different loop iterations by token edges, which stalls the computation pipeline as many cycles as the length of pipeline when assuming all operations have single cycle latency. In contrast, when token edges between a load and store operations are eliminated, the load operations feed the pipeline every cycle. Since the length of pipeline is 7, accurate memory dependence analysis allows a potential speedup of up to 7.

It is a difficult task to analyze memory accesses statically because the binary doesn't have high-level information, such as array, loop constructor, type, etc. In addition, some values are specified at runtime, for example, the size of image, the address of dynamically allocated object, etc. To overcome the problem mentioned above, we applied dynamic memory access analysis. As mentioned in Section 3.1, the static translator generates dependence test codes to analyze memory dependences at runtime. The dynamic translator gives runtime values to dependence test codes, and then they return memory access patterns. According to the result access patterns, the dynamic translator configures required token edges to synchronize dependent memory accesses. Section 4 describes the memory access analysis in detail.

3.4. Dynamic Binary Translator

The dynamic translator controls the reconfigurable hardware and test memory dependences by using dependence test codes given by the static translator. Figure 4 gives a block-view of the dynamic translator, which has a netlist cache to store netlists for hardware regions.

There are two kinds of overhead in dynamic translation: dependence test and token edge reconfiguration. Every time the dynamic translator executes the hardware region, it runs the dependence test code because it depends on runtime values. However, the test code is small such that it just computes few integer inequality equations and it is amortized by the loop execution. This overhead can also be reduced by running the hardware speculatively when input and output streams are allocated to different memory banks, respectively. The overhead of token edge reconfiguration occurs when the test code find out dependent memory accesses during the execution. In most kernels, dependences between memory accesses are fixed statically. For example, the input accesses and output access in **mult32** code are always independent, and two accesses in **fir** code always have loop-carried dependence between immediate loop

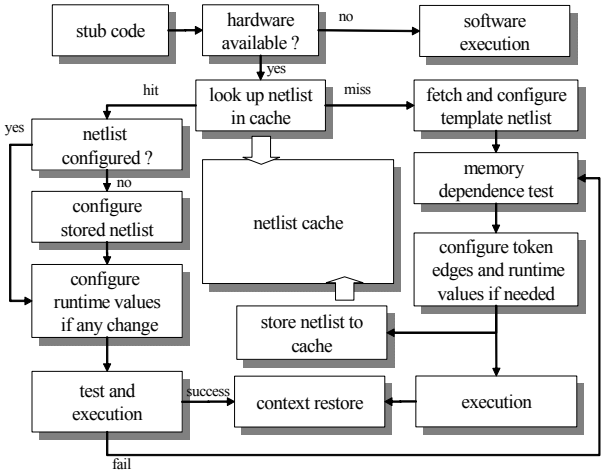


Figure 4 Dynamic binary translator framework

iterations. Therefore, the reconfiguration occurs only at the first execution in most kernels.

4. Memory Access Analysis

To prove that memory accesses in dataflow do not have loop-carried dependences, we must discover memory access regions accessed by memory operations. A memory access region is a set of memory locations accessed by a memory operation during the execution of a hardware region. In the example code of Figure 3–(a), access patterns are regular and monotonic; however, we do not have enough compile-time information to discover where the access region begins and ends in the memory space.

To cope with discovering memory access regions in the memory space, we make use of runtime values. In the example code, important values such as the base address of memory accesses and loop iteration count are constant during the loop execution, although those values are known at runtime. Values which remain constant in the certain region of code are called *runtime constants*.

4.1 Computing Runtime Constants

Computing runtime constants is similar to traditional constant propagation and runtime constant propagation on SSA [13]. It computes derived runtime constants for each operation of dataflow, propagating the initial set of runtime constants through the dataflow. The initial set contains symbolic values which represent values stored in registers and in the stack right before the entry of the hardware region. The derived runtime constant is an expression composed of constants, runtime constants, and data operators.

The dataflow in Figure 3-(d) has three derived runtime constants from the initial values of r7, r4, and r5. They

are represented as c(r4), c(r5), and c(r7) and correspond to the values of pointers and the loop count, respectively. In the rest of paper, both runtime constants and compile-time constants are called constants.

4.2. Recovering Memory Access Region

Compilers analyzing the memory access pattern in the source level use loop constructors and closed-form address expressions composed of induction variables and loop invariants. After the high-level code is compiled into the binary, such high-level information is spread on multiple instructions, but still remains without a significant loss of information. Accordingly, we can recover memory access patterns by using traditional analysis techniques, such as induction variable analysis, variable range analysis, and access pattern analysis.

To analyze induction variables, we applied Wolfe's analysis [14] and monotonic evolution [7]. The result of induction variable analysis produces the tuples of induction variables, each of which is defined to be *{IV, EV, STEP}*. Elements in the tuple are expression defined in constants, representing an initial value, an exit value, and an increasing or decreasing step, respectively. While the initial value is usually computed by constant propagation, the exit value is computed by range analysis. Range analysis performed on induction variables can be used to find the loop count. The range analysis propagates range information, focusing on branch instructions because induction variables as loop indices are bound by constant expressions at branch instructions. The exit value has the value *IV + STEP * loop count*.

By applying runtime constant propagation and induction variable analysis, we can obtain symbolic address expressions for memory accesses and convert each address expression into a sum-of-products form:

$$addr = \sum_{m=1} f_m(i_1, i_2, ..., i_n) + \sum Expr_C \qquad (1)$$

where $i_1, i_2, ..., i_n$ are induction variables. The function f is defined on induction variables. $Expr_C$ is a constant expression.

Equation (1) represents the access pattern of address expression over loop iterations. Many parallelizing compilers require the subscript of an array reference to be an affine expression of loop invariants and loop indices for analysis. In the similar reason, we require the function f to be an affine expression composed of induction variables and constants. If the function contains non-constant elements or it is not an affine expression, we call the term a *complex term*. Complex terms usually disturb analyzing memory access patterns, which decreases the amount of parallelism in the pipeline.

A memory access region has a tuple to represent its access range in the linear memory space. Given an address expression, $t_1 + t_2 + ... + t_n$ where t_i is a term in equation (1), an access range is defined to be a tuple of lower bound and upper bound:

$$(\sum_{k=1}^{n} \min(t_k), \sum_{k=1}^{n} \max(t_k)) \qquad (2)$$

where $\min(t_k)$ and $\max(t_k)$ are computed by initial values and exit values of induction variables.

4.3. Memory Access Cluster

In high-level languages, array references, each of which accesses a distinct array, are not always dependent on each other. Similarly, we classify memory accesses into several clusters. A cluster is defined as a set of memory accesses whose access ranges significantly overlap. In other words, memory accesses in a clusters access a common memory segment. The memory access region of a cluster is computed joining all memory access regions in the cluster. The access range of a cluster containing n memory accesses is

$$\forall k \in 0..n \quad (\min(LB_k), \max(UB_k)) \quad (3)$$

where LB_k and UB_k are lower bound and upper bound for *kth* memory access in the cluster. We classify memory accesses which have the same base expression symbolically into a cluster. Clustering begins by partitioning terms in equation (2) into two parts: *[base] + [offset]*. Our partition strategy in this paper is simple such that the terms of runtime constant expressions and f_k are classified into the base expression, and only compile-time constant expressions remain in the offset expression. This

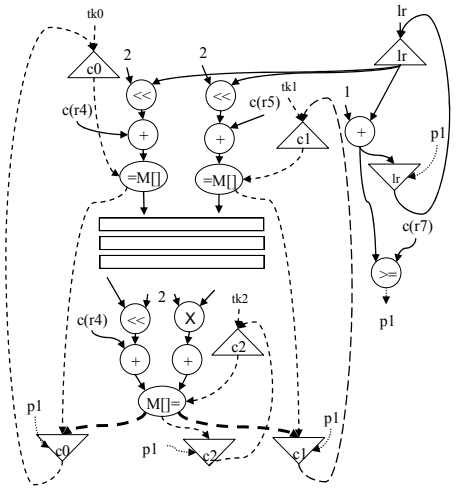


Figure 5 Dataflow graph after memory access analysis

approach generates small-sized clusters containing memory accesses whose addresses are shifted by compile-time constants, such as a[i] and a[i+1]. Large clusters require sophisticated array offset analysis, not only to test intra-cluster dependence, but also to test inter-cluster dependence.

Clustering reduces the overhead of the dependence test at runtime because the dynamic translator just tests whether clusters are dependent or not. For example, the first loop of the **idct** code has 16 memory accesses, but they can be classified into two clusters. Therefore, the dynamic translator calls the dependence test code just once.

Figure 5 shows the result of clustering. In contrast to Figure 3-(d), memory accesses are classified into three clusters, c0, c1, and c2 since memory accesses in the example code have distinct base expressions, such as *c(6) + (lr << 2), c(5) + (lr << 2), and c(4) + (lr << 2)*. Each cluster has its own mu and eta operations for parallel token flows. Intra-cluster token edges are represented by dashed lines, and inter-cluster token edges are represented by bold dashed lines.

To test if the three memory access regions for the clusters are dependent, the static translator generates two dependence test codes, each of which compares memory access regions between c0 and c2 and between c1 and c2. In the example dataflow, the dynamic translator detected that all clusters are not dependent. Inter-clusters token edges, therefore, were not configured.

5. Experiment Assessment

5.1. Experimental Setup

Our dynamic binary translator was executed on a simulator for the platform of an ARM processor and a reconfigurable architecture. All benchmarks were compiled by a GNU C compiler targeted to an ARM processor. Our simulator was implemented in SystemC to measure execution cycles on reconfigurable architecture. The ARM module runs the result binary of the static translator, and the reconfigurable architecture module simulates the Pegasus representation with realistic timing information, just as was done in [11]. The latency of ALU

Table 1 Benchmark information

Kernels	No. of ops
wvec	13
fir	11
fir2dim	44
n_complex_update	36
DSP_mult32_c	27
img_sobel_c	51
img_idct_8x8_c	633

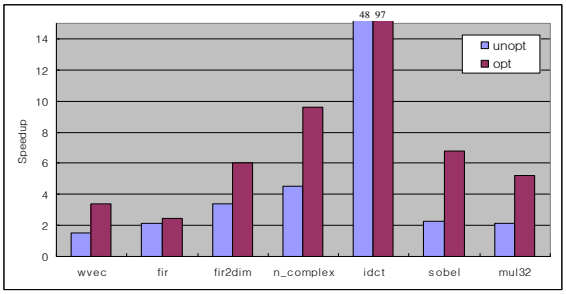


Figure 6 Execution times normalized to the software execution

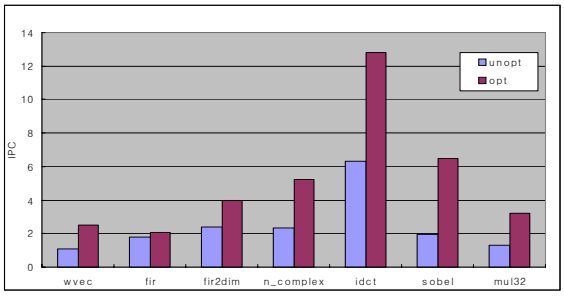


Figure 7 IPC (Instruction Per Cycle) for pipelines

operations are one cycle. The multiplier has three result latencies and one cycle issue latency. The hit latency of the cache is two cycles. Control operations have latency proportional to *log(number of inputs)*. Four memory accesses can be executed in a parallel fashion.

To compare the software execution and the hardware execution, we used a SimpleScalar-Arm because our ARM module is not cycle-accurate. The pipeline of SimpleScalar was configured to be similar to the pipeline of the ARM processor, and operation latencies are same as the reconfigurable architecture.

5.2. Experimental Results

This section reports the preliminary results of the memory access optimization on benchmark kernels. Table 1 shows the information of benchmark kernels from the DSPstone benchmark suite [10] and TI DSP/Image library [12]. They are typical signal processing applications with abundant parallelism. For **idct**, we used the first loop of two large-size loops. The second column is the number of primitive operations in the loop of binaries. The loop body of **idct** has a large number of operations because many local variables are used to store the result of computation temporarily, which leads the compiler to generate many stack access instructions.

Figure 6 shows the speedup of synthesized pipelines compared with the software execution. The bar labeled **unopt** describes the speedup of pipelines when the memory access optimization is not applied. All kernels

exhibit speedup compared to the software execution times because they exploit the fain grain parallelism in the dataflow. The result for **idct** is particularly interesting. Although **idct** contains a lot of operations in the binary, it shows a remarkable speedup. This is because stack access operations are eliminated in the preprocessing, and the computation pipeline has a high degree of parallelism.

The bar labeled **opt** shows the effectiveness of our memory access optimization. Speedup to **unopt** bars ranges from 1.73 for **fir2dim** up to 3.02 for **sobel**, except **fir**. The speedup of **fir** is 1.15 because the dynamic translator detected that two of the three memory access clusters have loop-carried dependence.

Figure 7 gives the IPC for synthesized pipelines. On all kernels, our optimization technique induced a higher instruction level of parallelism.

The overhead of reconfiguration occurs only at the first execution of fir. In other kernels, memory accesses are independent due to the characteristics of stream processing applications.

6. Conclusion and Future Work

This paper described dynamic binary translation and memory access optimization for reconfigurable architectures. Our memory optimization technique makes use of runtime information to overcome the limitation of static optimization. We presented encouraging results for signal processing kernels, inducing a promising speedup up to 3.02. The current limitation of our techniques is that complex terms could disturb the analysis of memory dependences; memory accesses for lookup tables often have complex address expressions. Future research should focus on overcoming the aforementioned limitations and examining more complex benchmarks.

7. References

[1] David Koes, Mihai Budiu, Girish Venkataramani and Seth Copen Goldstein, Programmer Specified Pointer Independence, *Workshop on Memory System Performance Washington*, DC, June 2004.

[2] Girish Venkataramani et al. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. *In International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2001.

[3] Greg Stitt and Frank Vahid. Hardware/Software Partitioningof Software Binaries. *In Internaltional Conference on Computer-Aided Design*, 2002.

[4] Greg Stittm, Roman Lysecky, and Frank Vahid. Dynamic Hardware/Software Partitioning: A First Approach. *In Design Automation Conference*, 2003.

[5] Yajun Ha, Bingfeng Mei, Patrick Schaumont, Serge Vernalde, Rudy Lauwereins, and Hugo De Man. Development of a Design Framework for Platform-Independent Networked Reconfiguration of Software and Hardware, *In International Workshop on Field Programmable Logic and Applications*, page 259, 2001.

[6] J-Y. Mignolet, S.Vernalde, D. Verkest, and R Lauwereins, Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances, *Proc. Of the 2nd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, p10-16, Las Vegas, Nevada, USA, June 2002.

[7] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua, Monotonic evolution: an alternative to induction variable substitution for dependence analysis, *Proceedings of the 15th International Conference on Supercomputing*, 2001.

[8] 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 Proceedings of the Conference on Programming Language Design and Implementation PLDI 1990*, pages 257–271, 1990.

[9] Mihai Budiu and Seth C. Goldstein, Optimizing Memory Accesses For Spatial Compuation, *In International Symposium on Code Generation and Optimization*, 2003.

[10] V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr, DSPstone: A DSP-Oriented Benchmarking Methodology, *Proc. of ICSPAT'94*, Dallas, Oct. 1994.

[11] Mihai Budiu, Spatial Computation, Ph D Thesis, *CMU CS Technical Report CMU-CS-03-217*, December 2003.

[12] Ti inc. http://dspvillage.ti.com.

[13] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad, Fast, Effective Dynamic Compilation, *In Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, USA, 1996.

[14] Michael Wolfe, Beyond induction variables, *In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 162-174, New York, 1992. ACM Press.