EVALUATOR-EXECUTOR TRANSFORMATION FOR EFFICIENT CONDITIONAL STATEMENTS ON CGRA

Yeonghun Jeong

Computer Engineering Program
Graduate School of UNIST

Evaluator-Executor Transformation for Efficient Conditional Statements on CGRA

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Yeonghun Jeong

06.20.2013
Approved by

[illegible]

Major Advisor
Jongeun Lee

Evaluator-Executor Transformation for Efficient Conditional Statements on CGRA

Yeonghun Jeong

This certifies that the thesis of Yeonghun Jeong is approved.

06.20.2013

[illegible]

Thesis Supervisor: Jongeun Lee

[illegible]

Won-ki Jeong: Thesis Committee Member #1

YChoi

Youngri Choi: Thesis Committee Member #2

Abstract

Control divergence poses many problems in parallelizing loops. While predicated execution is commonly used to convert control dependence into data dependence, it often incurs high overhead because it allocates resources equally for both branches of a conditional statement regardless of their execution frequencies. For those loops with *unbalanced* conditionals, we propose a software transformation that divides a loop into two or three smaller loops so that the condition is evaluated only in the first loop while the less frequent branch is executed in the second loop in a way that is much more efficient than in the original loop. To reduce the overhead of extra data transfer caused by the loop fission, we also present a hardware extension for a class of coarse-grained reconfigurable architectures (CGRAs). Our experiments using MiBench and computer vision benchmarks on a CGRA demonstrate that our techniques can improve the performance of loops over predicated execution by up to 65%, or 38.0% on average when the hardware extension is enabled. Without any hardware modification, our software-only version can improve performance by up to 64%, or 33.2% on average, while simultaneously reducing the energy consumption of the entire CGRA including configuration and data memory by 22.0% on average.

Contents

List of Figures

CHAPTER I

Introduction

The need for higher energy efficiency across all domains of computing from embedded to supercomputers is the main driver behind the increasing popularity of accelerators, which include applications specific hardware as well as more programmable ones such as reconfigurable architectures and GP-GPUs. For those architectures, one important source of energy efficiency is Loop-Level Parallelism (LLP), and pipelining is the preferred way of exploiting LLP on reconfigurable architectures, which have abundant, possibly heterogeneous, processing elements. However except for a few application domains such as DSP (Digital Signal Processing), even loops can be quite complicated, and thus pipelining a loop optimally for a Coarse-Grained Reconfigurable Architecture (CGRA) is an area of active research [5,11,12,16,22].

One important problem in this context is how to handle conditional statements inside loops. Most typically, conditional statements are handled using *predicated execution* [18,21]. However, predicated execution is wasteful because it allocates resources for both branches of a conditional, even if, quite certainly, only one of them is to be executed. To address this problem of wasted resources, a technique called Dual-Issue Single-Execution (DISE) [12] was proposed for a CGRA. The key idea is to issue operations from both paths simulateneously, but at runtime, execute only one of them depending on the prediate value. This can save cycles and energy, but unfortunately becomes less effective when the sizes of the branches are unbalanced, which is often the case in applications.

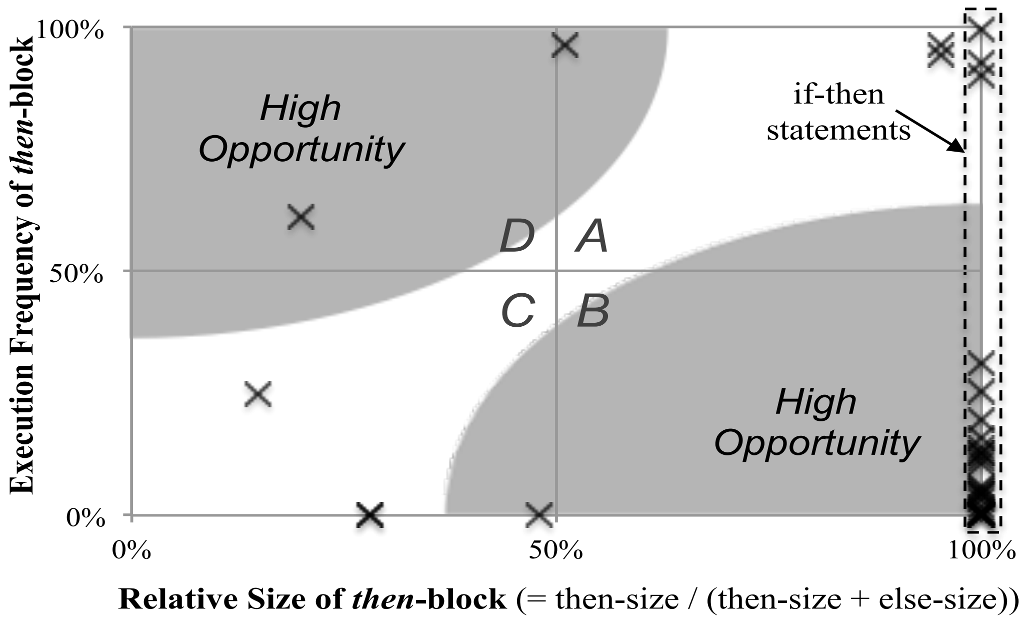


Figure 1.1: Frequency and size of then-blocks in loops from embedded and vision benchmarks. Conditional statements in loops have highly unbalanced branches in terms of both size and execution frequency. Of special insterest are the then-blocks in loops of region B and the else-blocks in loops of region D, which are large but rarely executed.

We have examined conditionals[1] inside loops from applications in MiBench [10] and a computer vision benchmark suite [26], in terms of both the relative size and execution frequency of the then- and else-blocks. The results are plotted in Fig. 1.1, where the x-axis represents the size balance and the y-axis the execution frequency balance. The center of the graph thus represents the perfect balance whereas the right (left) side is where then-blocks (else-blocks) are larger, and the top (bottom) side is where then-blocks (else-blocks) are more frequently executed.

This graph reveals several useful insights. First, the majority (76%, or 25 out of 33) of the conditionals inside loops have then-blocks only (no else-blocks). Even the rest of them are mostly unbalanced, leaving only two loops near the 50% line in terms of size. This suggests that most loops at least in the applications we have examined have quite unbalanced conditionals, rendering techniques like DISE unattractive. Second, from the vertical distribution we see that most conditionals are very polarized, strongly favoring one branch over the other. This has an important implication that while predicated execution is very resource-efficient for the if-then statements in region A because there then-blocks are almost always executed, for those in region

[1]These do not include loop control branch instructions but only those explicitly stated in the source code.

B, it is very wasteful, presenting an opportunity for improvement over predicated execution. Third, interestingly enough, much more loops are found in region B than in region A, pointing to the need for addressing the problem. Thus our goal in this paper is to develop a technique that improves the efficiency of loops belonging to regions B and also D due to the symmetry of the axes. Our technique is complementary to predicated execution, which is already efficient for loops in regions A and C.

In this paper we propose for CGRAs a compiler technique that basically divides a loop with conditionals into two (or three) smaller loops, where the first loop only evaluates the condition while the second one executes the then-block, only for thoes iterations where the condition is true. This is very similar to loop fission, but we *compress* the second loop for more efficient execution. We call this *evaluator-executor transformation*, and its performance and energy improvement comes from the fact that it can bypass unnecessary iterations completely, not wasting any energy or resources for them. There are some challenges however. First, while splitting a loop is rather straightforward for those with only one if-then statement, it is not obvious how to handle if-then-else or other more complex conditionals including nested and/or a series of conditionals. Second, splitting a loop in this way can break the live range of some scalar variables, which then need to be passed from one loop to another. This is a by-product of loop fission and may offset the performance and energy advantage of our technique, requiring careful optimization. Lastly, loops with inter-iteration data dependence (recurrent loops) may not be easily fissioned. We present two techniques—one is a pure software technique and the other is based on hardware extension—to address those challenges.

Our experiments using loops with conditionals from MiBench and computer vision benchmarks demonstrate that our hardware technique can improve the performance of loops with conditionals over predicated execution, by up to 65%, or 38.0% on average across applications. Our software-only version, which does not require any hardware modification, can improve the performance over predicated execution by up to 64%, or 33.2% on average. While our software transformation generates loops that use more memory operations than the original, thereby increasing the dynamic energy in the memory, the increase is very modest, and in many cases much less than what we save on the static energy through runtime reduction. Overall our software technique can reduce the energy consumption of the entire CGRA including configuration and data memory by 22.0% on average.

CHAPTER II

Related Work

Our target architecture, CGRA [14], is a type of reconfigurable architecture where each processing element (PE) is at the granularity of words rather than bits as in FPGAs. This has both advantages and disadvantages. The advantages include fast runtime reconfiguration, high efficiency for operations directly supported by the PE (e.g., arithmetic), and lower programming barrier for software engineers with no hardware design background, while the disadvantages include inefficiency in bit-level manipulation or in general any operation not directly supported by the "instruction set" of a PE, and the lack of flexibility in terms of trading off the cycle time for easier application mapping. There are various ways of mapping loops onto CGRAs [4, 17], but loop pipelining, or software pipelining-based mappings [20, 22] can effectively utilize heterogeneous PEs yielding high-throughput schedules, and can also handle recurrent loops. On the other hand, for parallel loops on homogeneous PEs, SIMD (Single-Instruction Multiple-Data) can lead to higher utilization because no PE is wasted for *routing* unlike in software pipelining.

Control divergence poses many challenges when mapping loops for CGRAs. Software pipelining-based mappings typically use predicated execution [23], which converts control flow into data flow, by issuing instructions from both branches of a conditional (thus no control dependence) but executing them only if the predicate is true (thus data dependence). Predicated execution requires hardware support such as (i) adding predicated versions to some (*partial predication*) or all (*full predication*) instructions, (ii) a predicate register file, which may be shared among all PEs, and (iii) one or more predicate-defining instructions that can write to the predicate register file.

Recent work for CGRAs tries to improve the efficiency of control flow in loops. DISE (Dual-Issue Single-Execution) [12] issues two instructions simultaneously, but executes only one of them with the "true" predicate, which can increase performance especially when the conditional has balanced branches. SFP (State-based Full Predication) [13], on the other hand, can achieve energy saving by putting a PE into a low power mode if the PE is to recieve operations that will be disabled due to predication. While SFP has been applied to mappings that are not pipelined, SFP for pipelined mappings is not known.

BERET (Bundled Execution of Recurring Traces) [8] is an accelerator technique that exploits the existence of a *common scenario* in a control flow. However, BERET does not fission loops, but instead restarts an iteration on the main processor whenever control diverges from the common scenario, which is monitored by the BERET hardware. Again, restarting an iteration is difficult if the loop is pipelined. Also, CGRAs have a larger overhead for switching to the main processor, making it far less efficient if BERET were applied to CGRAs.

GPUs also suffer from control divergence in loops, which can be addressed by dynamically forming SIMD-instructions from large collections of threads [7, 24]. Simultaneous Branch and Warp Interweaving [2] executes operations from divergent control paths or even from different scheduling groups (warps) *simultaneously* as in the same SIMD instruction. For unstructured control flows, finding out when divergent threads reconverge can be important and challenging, which is addressed by [6].

In high-level synthesis, previous work handles conditional statements in pipelined loops using techniques like path-based scheduling algorithms [3], speculation techniques [9], or static branch prediction [15].

Our work is based on loop fission [1], a well known loop transformation technique. Our technique specializes loop fission for handling control divergence, and extends it by compressing the second loop, which is the main source of performance and energy improvement.

CHAPTER III

Evaluator-Executor Transformation

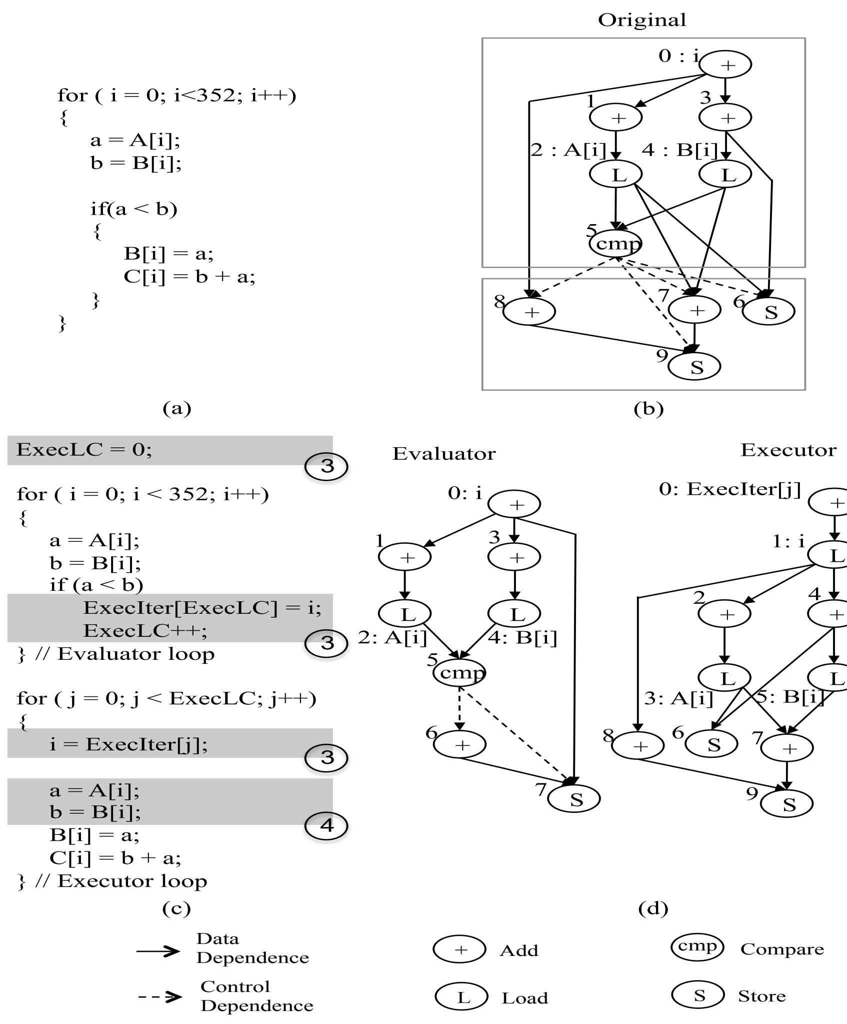
3.1 Basic Idea

The basic idea of our evaluator-executor transformation is to divide a loop into smaller ones, such as one that includes operations before and up to the condition evaluation, and one that includes operations after the condition evaluation. The evaluator loop also stores the iterator values for the iterations where the condition holds true, and the iterator values are later retrieved in the executor loop. Improvement of performance and/or energy is the main motivation, as the executor loop often has much less trip count than the original loop.

3.1.1 Illustrative Example

Fig. 3.1 illustrates a simple loop with an if-then statement. The control flow graph of the loop body has only two basic blocks. The condition expression is evaluated in the first block, and only when the condition is true, the second basic block is executed.

In the conventional mapping based on predicated execution only, the control dependence is converted into data dependence, creating data dependence edges from 5 to 6 $\sim$ 9 in Fig. 3.1(b). These additional edges may seem to complicate routing for mapping, but the hardware architecture may provide special support for routing of predicate values, which are only one-bit signals.



```
for ( i = 0; i<352; i++)
{
    a = A[i];
    b = B[i];

    if(a < b)
    {
        B[i] = a;
        C[i] = b + a;
    }
}
```

```
ExecLC = 0;

for ( i = 0; i < 352; i++)
{
    a = A[i];
    b = B[i];
    if (a < b)
        ExecIter[ExecLC] = i;
        ExecLC++;
} // Evaluator loop

for ( j = 0; j < ExecLC; j++)
{
    i = ExecIter[j];

    a = A[i];
    b = B[i];
    B[i] = a;
    C[i] = b + a;
} // Executor loop
```

Figure 3.1: Splitting a loop with an if-then statement into evaluator and executor loops. (a) Original loop, (b) control-data flow graph of the original loop, (c) evaluator and executor loops, and (d) their data flow graphs.

The main weakness of predicated execution, however, is the waste of resources allocated to the predicated block (the second block in our example), which really needs to be executed only when the condition is true. But since the condition value changes from iteration to iteration, and the loop schedule must be generated statically, resources must be allocated for the second block regardless of the condition value, which leads to waste of resources.

Instead we create two loops, namely evaluator loop and executor loop. Essentially, the evaluator loop contains the code up to the condition evaluation, and the executor loop contains the code below the condition expression. To preserve the functionality, a few operations need to be added, such as saving and recalling of the iterator values for true iterations.

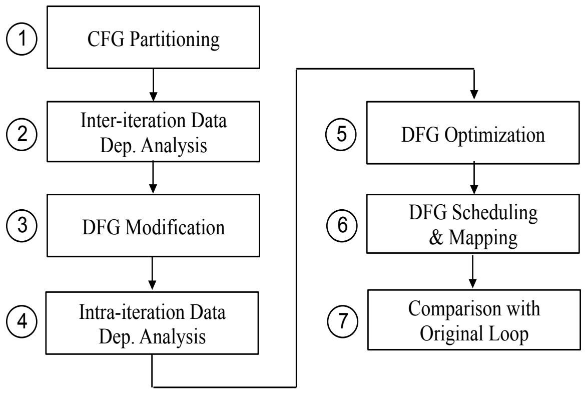

1
CFG Partitioning
2
Inter-iteration Data Dep. Analysis
3
DFG Modification
4
Intra-iteration Data Dep. Analysis
5
DFG Optimization
6
DFG Scheduling & Mapping
7
Comparison with Original Loop

Figure 3.2: The overall flow of our evaluator-executor transformation.

3.1.2 Challenges

While the transformation may seem trivial for the previous example, the control structure in the loop body in general may be much more complicated. Also if there is a statement after the conditional, simple two-loop fission will not be enough. Thus the first challenge is, given an arbitrary control flow graph of a loop body, to determine how to partition the basic blocks to fissioned loops so that the functionality is preserved and the benefit of the loop fission is maximized.

Recurrent loops, or loops with inter-iteration data dependence, pose another challenge, as data dependence cycles should not be broken during loop fission. In some cases we can circumvent data dependence cycles and still perform our loop transformation.

Another challenge is intra-iteration data dependence such as a scalar variable defined before, but used inside, a conditional statement. Such variables need to be passed to the executor loop, which requires adding new operations to evaluator and/or executor loops.

3.2 Overall Flow

Fig. 3.2 illustrates the overall flow of our evaluator-executor transformation. The control flow graph (CFG) of the loop body is partitioned into three sets, which we call simply A, B, and C, where set C may be empty. They correspond to evaluator, executor, and trail loops, respectively. Next, inter-iteration data dependence analysis is performed to see if there is recurrence cycle(s) that cannot be handled. Some loops with recurrence cycles can be successfully fissioned; for others, our transformation fails. Then the blocks included in each set is converted into data flow graphs, which are modified by adding operations necessary for our transformation. The next step is intra-iteration data dependence analysis, which is to identify the variables that need to be passed from the evaluator to the executor or the trail loop. Such (temporary) variables may be passed through the local memory or simply recomputed—we decide it by comparing their estimated cost. The next step, DFG optimization, cleans the DFG for any possible dead code introduced during the previous steps (e.g., recomputing a temporary variable in the executor loop may make the original definition in the evaluator loop unnecessary). Finally the DFGs are scheduled and mapped onto the target architecture, and the expected runtime is compared with that of mapping the original loop using predicated execution only, where the better is selected.

We now explain the key steps of the flow, starting with the CFG partitioning.

3.3 CFG Partitioning

The problem of CFG partitioning is to determine which basic blocks to include in each fissioned loop, which is far from trivial for an arbitrary control flow graph. This problem arises even for a simple if-then-else statement because the executor loop cannot contain both the then- and else-blocks, which defeats the purpose of our transformation.

Before we proceed, let us clarify one underlying assumption here. To efficiently support an arbitrary control flow graph, or even a simple if-then-else statement in a loop, we assume that the architecture supports predicate execution. Otherwise we have to create a loop for each of the then- and else-blocks, which incurs much more overhead, and is likely to cancel out any performance advantage of our transformation. With this architecture, any of the fissioned loops can contain control statements inside.

3.3.1 Performance Consideration

Consider a simple case where the loop contains only one if-then-else statement without any statement afterward, as illustrated in Fig. 3.3(a). Let us assume that the branch probability is p, and the then- and else-blocks take S_T and S_E cycles on average, respectively (which may be approximated to the number of operations in each block). Further, the loop iterates N_{orig}

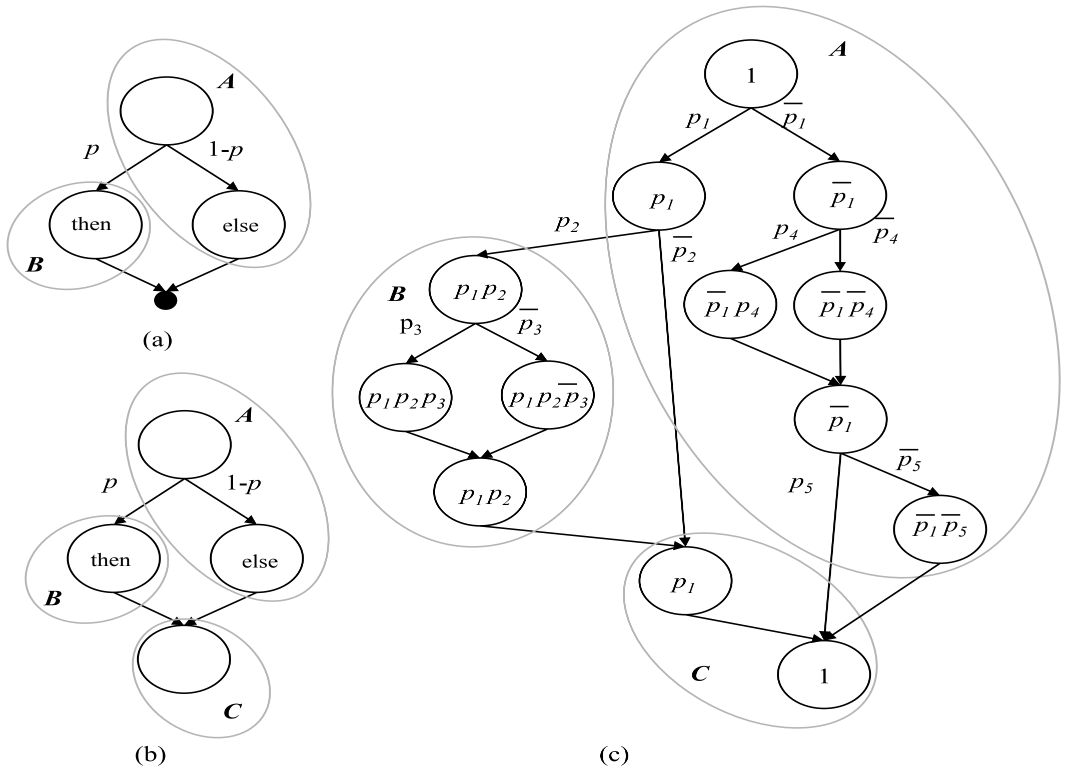


Figure 3.3: Various control flow graphs.

times, and the condition block takes S_C cycles on average.

Depending on which block is included in the executor loop, the expected runtime after loop fission can be represented as follows, assuming that the runtime of two blocks executing together is the sum of the runtime of each running separately:[1]

$$\begin{aligned} T_{\textit{then-in-executor}} &= (S_C + S_E)N_{orig} + S_T N_{orig} \cdot p \\ T_{\textit{else-in-executor}} &= (S_C + S_T)N_{orig} + S_E N_{orig}(1-p)\,, \end{aligned} \tag{III.1}$$

from which the condition for choosing the then-block is: $\frac{S_T}{p} > \frac{S_E}{1-p}$. In other words, the best candidate for the executor loop is a large and rarely executed one. We generalize this intuition for our partitioning algorithm, which first identifies all the candidate blocks or sets of blocks, and chooses the one that maximizes the size-over-execution-frequency metric.

[1]This assumption holds true for a RISC processor and is also valid when the runtime is approximated as the number of operations.

```
if condition1 then
    if condition2 then
        Block ❶ (Executor)
    else
        Block ❷
    end if
else
    if condition3 then
        Block ❸
    end if
    Block ❹
end if
Block ❺
```

Figure 3.4: Determining the set of blocks for the trail loop.

3.3.2 Trail Loop

A slightly more complex case is one that has statements after a conditional, as illustrated in Fig. 3.3(b). Such statements can have dependence on the conditional, especially on the blocks included in the executor loop. In this case, the post-conditional statements must be executed after the executor loop, as a sepearate loop. This loop is called trail loop.

In a nested control statement, determing which blocks should be included in the trail loop may not be obvious. For instance, in Fig. 3.4 if the executor includes block ❶ only, the trail loop does not include blocks ❷, ❸, or ❹, but block ❺ only. The principle here is that all the blocks that may be executed in an iteration where the executor block is executed must be included in the trail loop if they are executed after the executor block. Based on this observation, we provide an algorithmic solution for the general case in the next section.

3.3.3 General Solution

Our solution for an arbitrary CFG generated from a structured code (i.e., without gotos) is as follows: (i) identify all the candidate sets of blocks, and (ii) choose the one that maximizes our fitness metric (size over execution frequency). In the discussion below, we assume without loss of generality that every conditional statement has up to two cases—if-then or if-then-else statements only. First we associate a unique predicate variable with each condition expression as illustrated in Fig. 3.3(c). Then we can find the predicate expression that dictates the condition for executing each block in the CFG, which is a product of predicate variables or their complements, as shown inside each block in the figure. Let us call this product the *execution condition* of the block.

Let B denote the set of all blocks included in the executor loop. Then we note that if B includes a block whose execution condition is x, adding to B all the other blocks whose execution condition contains x can only increase the benefit of the executor, as per our performance model. This is because adding them does not increase the execution frequency of B but only increases its size and runtime. This is in effect saying that in a nested conditional statement, all the inner conditionals must be included in their entirety if the outer one is included. In our example, these are all the candidate sets of blocks (each set represented by its maximal execution condition): $\{p_1,\ p_1p_2,\ p_1p_2p_3,\ p_1p_2\overline{p_3},\ \overline{p_1},\ \overline{p_1}p_4,\ \overline{p_1}\cdot\overline{p_4},\ \overline{p_1}\cdot\overline{p_5}\}$. Finding all the candidates can be done efficiently.

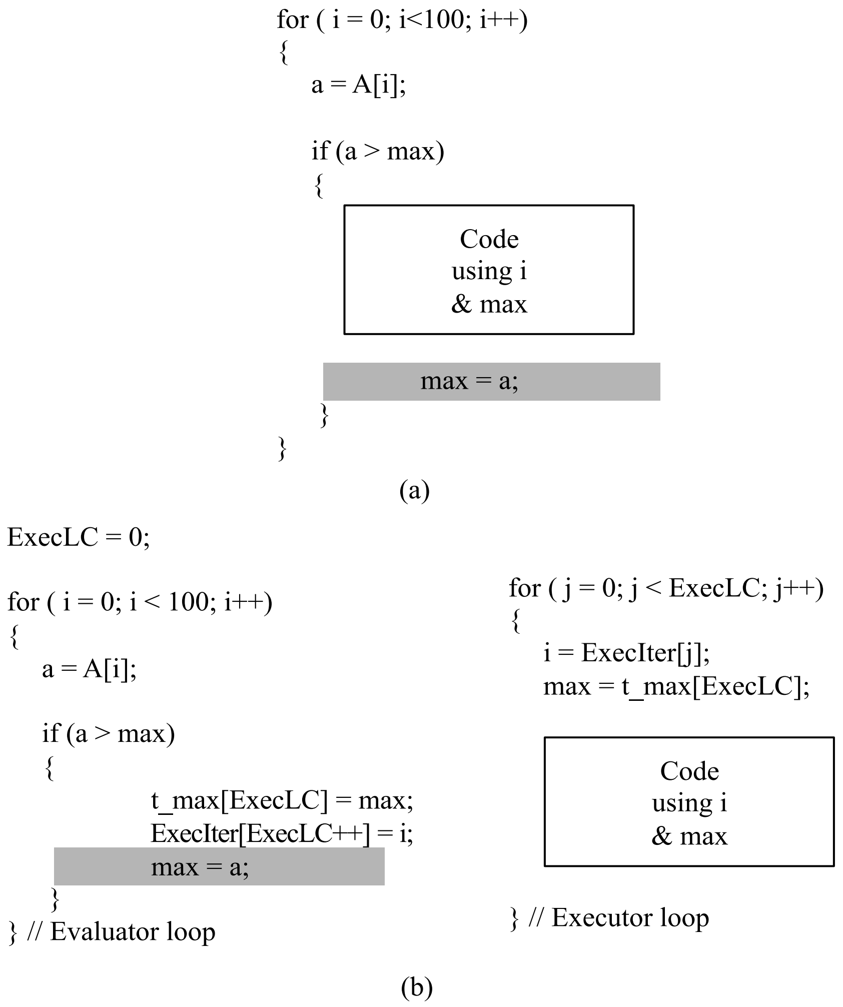
The set of blocks included in the trail loop, called C, includes all the blocks that (i) appear after the blocks of B in the original code, and (ii) whose execution condition y is *not* exclusive with that of B, i.e., $y \cdot x \neq 0$, where x is the union of all the execution conditions of the blocks in B. Fig. 3.3(c) shows set C for the B selected as in the figure.

All the remaining blocks are included in set A, which is for the evaluator loop. In a later step (i.e., intra-iteration data dependence analysis), if the operations in C are found to have no data dependence at all on B, C can be merged into A.

3.4 Inter-iteration Data Dependence

In a loop, an operation that is dependent on the same or another operation of a previous iteration, either through a scalar or an array variable, creates a cycle in the data dependence graph. Since data dependence represents precedence relationship, such cycles may not be broken into different loops, which often complicates loop fission. One caveat, however, is that a data dependence cycle that remains completely in one of the fissioned loop does not interfere with loop fission. Therefore, we try moving operations between the sets (e.g., from A to C, or from C to A)[1] in order to make a data dependence cycle completely contained in one set. Fig. 3.5 illustrates this for a loop with the max operation. The max operation was inside the

[1]Moving operations from B to A or C is possible but needs to be predicated. It is impossible to move operations from either A or C to B.



```
for ( i = 0; i<100; i++)
{
    a = A[i];

    if (a > max)
    {
        Code
        using i
        & max

        max = a;
    }
}
```

(a)

```
ExecLC = 0;

for ( i = 0; i < 100; i++)
{
    a = A[i];

    if (a > max)
    {
              t_max[ExecLC] = max;
              ExecIter[ExecLC++] = i;
              max = a;
    }
} // Evaluator loop
```

```
for ( j = 0; j < ExecLC; j++)
{
    i = ExecIter[j];
    max = t_max[ExecLC];

    Code
    using i
    & max

} // Executor loop
```

(b)

Figure 3.5: Recurrent loops can be fissioned if the recurrence can be isolated into one fissioned loop. This example illustrates that by moving an operation (in the gray box) outside of the conditional, we can break the dependence of the evaluator loop on the executor.

conditional in the original loop but is now taken outside of it, thus creating an evaluator loop that is no longer dependent on the executor loop. If all the data dependence cycles are contained by individual sets, we can proceed to the next step. Otherwise the evaluator-executor transformation is aborted.

3.5 DFG Modification

Our transformation requires the evaluator loop to store two pieces of information: the iterator values for the *true* iterations and the total number of *true* iterations. This can be done by adding a few operations in the DFG of set A. While the evaluator and the trail loops inherit the loop control code (i.e., iterator initialization and increment) of the original loop, the executor loop has a trivial loop control code, and inside the loop body, the real values of the original iterator is retrieved from the memory. Thus one load operation and related operations (for address calculation) need to be added to set B.

3.6 Intra-iteration Data Dependence

An example of intra-iteration data dependence is a variable defined in set A and used in B or C. Such variables, which we call *transfer variables*, may not be available in B or C unless they are explicitly copied. One can either save and restore the transfer variable through the local memory after first converting it into an array (similar to *variable expansion* in software pipelining [23]), or simply recompute the variable where it is needed. We try both and choose the better one, or the one that requires fewer additional operations, possibly with some weighting factors. In the case of save-and-restore, a transfer variable used in B only, needs to be saved for *true* iterations only. In the case of recomputation, the expression tree corresponding to the definition of the variable needs to be copied from A.

A more complicated case is when a variable is defined in A, modified in B, and used in C. Such variables can be passed through the local memory, which is similar to the simpler case. Recomputing them in C can be tricky however, because it involves copying both its definition (from A) and the conditional update (from B).

CHAPTER IV

Fission-Specialized CGRA

While the previous section presented our evaluator-executor transformation in a largely target architecture-independent manner, we now propose a hardware extension that is specially tailored for the transformation, for a class of coarse-grained reconfiguration architectures (CGRAs). This architecture extension also accommodates loop fission for CGRAs.

As the extra code in our loop transformation originates from:

1. saving and restoring iterator for true iterations,
2. counting true iterations, and
3. either save-and-restore or recompute transfer variables,

our strategy is (i) to have a hardware counter for counting true iterations, and (ii) to provide efficient means of saving and restoring iterator and transfer variables.

4.1 Hardware for True Iteration Counting

Fig. 4.1 illustrates the architecture extension for a quadrant of a 4x4 CGRA. The baseline CGRA is very similar to the ADRES architecture [19]. The hardware for counting *true* iterations is very simple, consisting of one counter only (True_CNT in the figure), which serves the entire CGRA. The *true* counter is driven by the OR-ed result of all XE (Executor Enable)

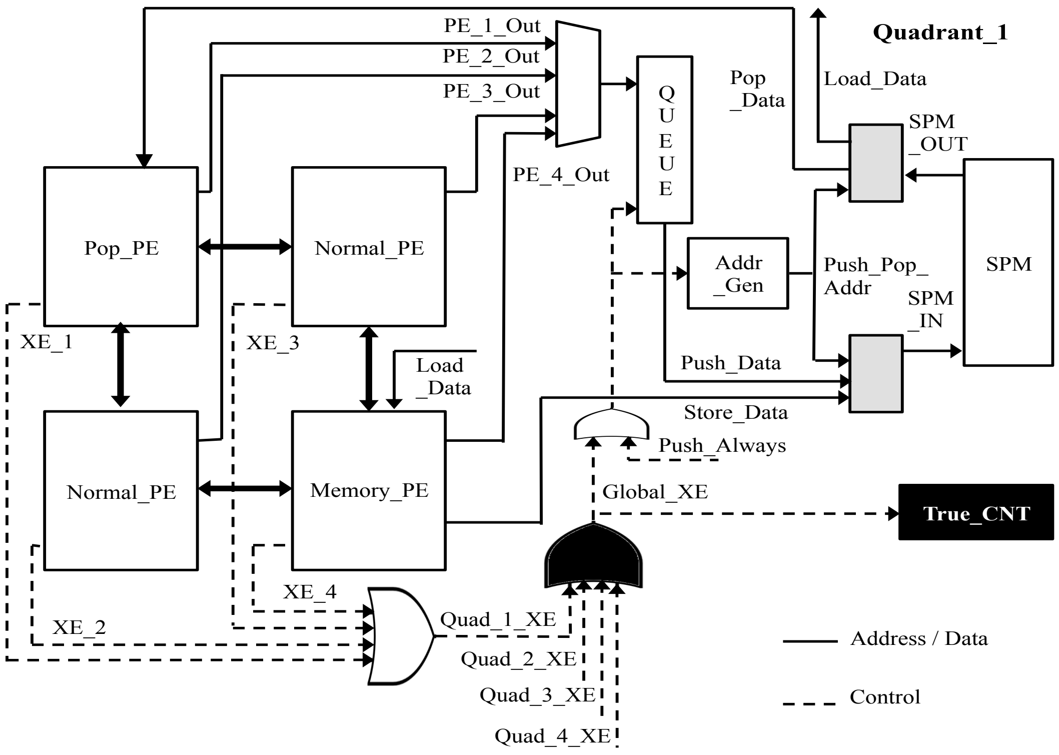


Figure 4.1: Extending each quadrant (2x2 PE array) of the baseline CGRA with: a FIFO queue, an address generator, XE (Executor Enable) output signals, *pop* operation in one PE, and *push* operation in every PE. A *true* iteration counter is also added to the CGRA. Black-filled components are unique in the entire CGRA. SPM (ScratchPad Memory) is the CGRA's local memory.

outputs of PEs (Processing Elements). The XE output of a PE is identical to its predicate output, except that it is driven high only when the PE is actually computing the condition value in an Evaluator loop; thus, usual predicated execution does not interfere with true iteration counting. The CGRA's controller is aware of the *true* counter, and its final value, which is the total number of true iterations, is used as the trip count of the Executor loop.

4.2 Hardware for Save and Restore

4.2.1 Push and Pop

To save and restore the iterator or a transfer variable, a number of operations may be needed, including a store/load operation and the accompanying address calculation computation. What they do, however, is simple push and pop operations using the CGRA's local memory as a FIFO.

Thus we simplify this by replacing the whole address calculation with a very simple address generator with no multi-threading, which can be implemented with a counter, and by embedding the store operation into other instructions as a sub-operation controlled by one bit of the configuration.[1] This sub-operation is called *push*, and therefore can be executed by any PE. The load operation is not embedded, but provided as a separate instruction (called *pop*), which is simpler than a load because the address need not be specified.

4.2.2 Optimizing for Performance

To explain the asymmetrical design decision between push and pop, let us consider the runtime of a software-pipelined loop on a CGRA, which can be modeled as $II \cdot (L + N - 1)$, where *II*, L, and N are the initiation interval, the number of stages (the schedule length in terms of *II*), and the number of iterations, respectively. Then the runtime difference between the original loop and the fissioned loops is given as follows (assuming no trail loop):

$$\begin{aligned} T_{orig} &= II_{orig}(L_{orig} + N_{orig} - 1) \\ T_{fission} &= II_1(L_1 + N_{orig} - 1) + II_2(L_2 + pN_{orig} - 1) \\ \Delta T &\approx (II_{orig} - II_1 - pII_2)N_{orig}\,, \end{aligned} \tag{IV.1}$$

where the subscripts, viz., *orig*, 1, and 2, denote the original loop, the evaluator, and the executor, respectively, and p is the executor's execution ratio. The approximation in the last equation is based on the assumption that $N \gg L$ in general. From the last equation it is clear that to maximize the difference, we must minimize II_1, or the initiation interval of the evaluator.

We design *push*, which is used in the evaluator, such that it requires no additional instruction (thus no additional PE) and it can be performed on any PE (no pressure on placement or routing). On the other hand, *pop* is designed to require an instruction (one PE needed) and it must be done on a particular PE in each quadrant. Luckily the pressure on routing due to pop is not high, because a pop operation is usually a leaf node of an expression tree, and therefore has much freedom in terms of placement and routing anyway.

Each quadrant supports one push/pop operation per cycle, and thus a CGRA can access up to four variables including an iterator using push/pop in any loop. If more variables need

[1] This requires reserving one bit per PE in the configuration for the push operation.

to be transferred, we do it in software, allocating hardware to the earliest appearing variables.

4.2.3 Queue

Another peculiarity of the push operation is that it may[1] need to be done only in true iterations, or when the condition expression is true. Often there is a time lapse between the definition of a variable and the availability of the condition value, the latter of which is usually at the last stage. The worst case is the iterator, which is typically defined at the first stage. In such a case, we can let the variable or the iterator stay in the PE or keep routing it among PEs, both of which invariably waste resources and add to the routing pressure. Thus we add a queue to hold such temporary data until they are either written to the local memory or discarded. The decision of whether to write or discard the data is given by the global version of the XE signal (see Fig. 4.1). This queue is updated once every initiation interval by the CGRA's controller.

From the worst case analysis, the queue size only needs to be as large as the number of stages. Our experimental results using embedded and vision applications indicate that the maximum number of stages is 5.5 on average across applications, and up to 11 for two applications. Thus we use a 12-entry queue. Now, this fixed queue size does not mean that certain loops cannot be mapped if they happen to have more stages than the queue size, but it only means that the mapping has to be changed, such that either the number of stages is reduced possibly at the expense of the increased initiation interval, or some variables should wait in PEs during the excess stages.

In summary, our architecture extension is designed to nearly eliminate the code overhead of our evaluator-executor transformation for a modest amount of hardware, as listed below.

- (Per PE) adding push operation
- (Per quadrant) adding pop operation to one of the PEs; a FIFO; and a counter for address generation
- (Per CGRA) a counter for counting true iterations.

[1]Some variables, such as one that is used in the trail loop, may need to be pushed every iteration regardless of the condition value (see Push_Always in Fig. 4.1).

CHAPTER V

Experiments

5.1 Experimental Setup and Applications

To evaluate the effectiveness of our proposed technique, we use a CGRA similar to the ADRES architecture [19], which supports predicated execution, and consists of a 4x4 heterogeneous PE array and a multi-banked local memory. Our CGRA has four load-store PEs, one in each row, that are connected to the four banks of the local memory, where each is a 256 KB scratchpad. A load operation from a PE takes 6 cycles due to a crossbar switch between the load-store PEs and the memory banks. The interconnection between the PEs includes mesh and diagonal. Every PE can perform ALU operations while more expensive operations (e.g., multiplication) can be done by four specific PEs only. Operations on a PE are fully pipelined, and can be predicated. There is no central register file except for a predicate register file, which can be updated by any PE. For mapping, we use modulo scheduling based on the EMS algorithm [22]. For our hardware extension, we use a 12-entry queue in each quadrant.

We use applications from MiBench [10] and SD-VBS [26], which a computer vision benchmark. We use all the loops with at least one conditional that can be mapped to our CGRA. Loops that have less than 16 iterations or those that cannot be split due to inter-iteration data dependence (even after moving operations around) are excluded, finally resulting in a total of 11 loops. Table V.1 lists the characteristics of the loops. The number of operations on Column

Table V.1: Kernel characteristics

Application	#Blocks	#Op	f_{ex}	r_{ex}	V
Cjpeg	2	8	4%	25%	1
Dijkstra	3	21	5%	38%	2
Lame_1	2	28	93%	64%	1
Lame_2	3	27	94%	59%	4
Lame_3	3	14	0%	36%	2
Susan_edges	2	216	31%	2%	2
Tiffdither	18	40	6%	13%	4
Tiffmedian	9	41	2%	63%	4
Disparity	2	11	16%	36%	2
Multi_ncut	2	11	0%	45%	2
Stitch	2	14	4%	57%	2

Note: f_{ex} and r_{ex} represent the frequency and the relative size (compared to the entire loop body) of the executor loop, respectively, and V is the number of iterator/transfer variables.

3 includes only the ones included in the DFG of the original loop, but not the loop-control related ones, which are taken care of by the CGRA controller. The 4th column shows how often the executor loop is executed per iteration on average (i.e., execution frequency), and the 5th column is calculated as the ratio of the number of operations in the executor loop to that of operations in the entire loop body, before applying evaluator-executor transformation. The last column is the number of variables including the iterator that need to be transferred between loops.

5.1.1 Performance of Our Transformation with Hardware Extension

For performance evaluation, we compare three cases in this section. First we map each loop using predicated execution only, denoted by **P**, which is the baseline of our comparison. Then we apply our evaluator-executor transformation with *hardware extension enabled*, which is denoted by **H**. We also estimate the number of *dynamic* operations that would be executed if the block(s) corresponding to the executor loop were executed only when necessary while the other blocks were executed in every iteration, denoted by **Op** in the graph. We use this formula, $(1 - r_{ex}) \cdot 1 + r_{ex} \cdot f_{ex}$, where r_{ex} and f_{ex} are the size ratio (in the number of operations) of the block(s) selected for the executor loop and their execution frequency, respectively (i.e., the 4th and 5th columns of Table V.1). This metric represents the minimal number of operations that must be executed in any loop fission-based scheme including ours without considering any overhead of loop fission, and therefore may reveal how effectively our technique can realize the

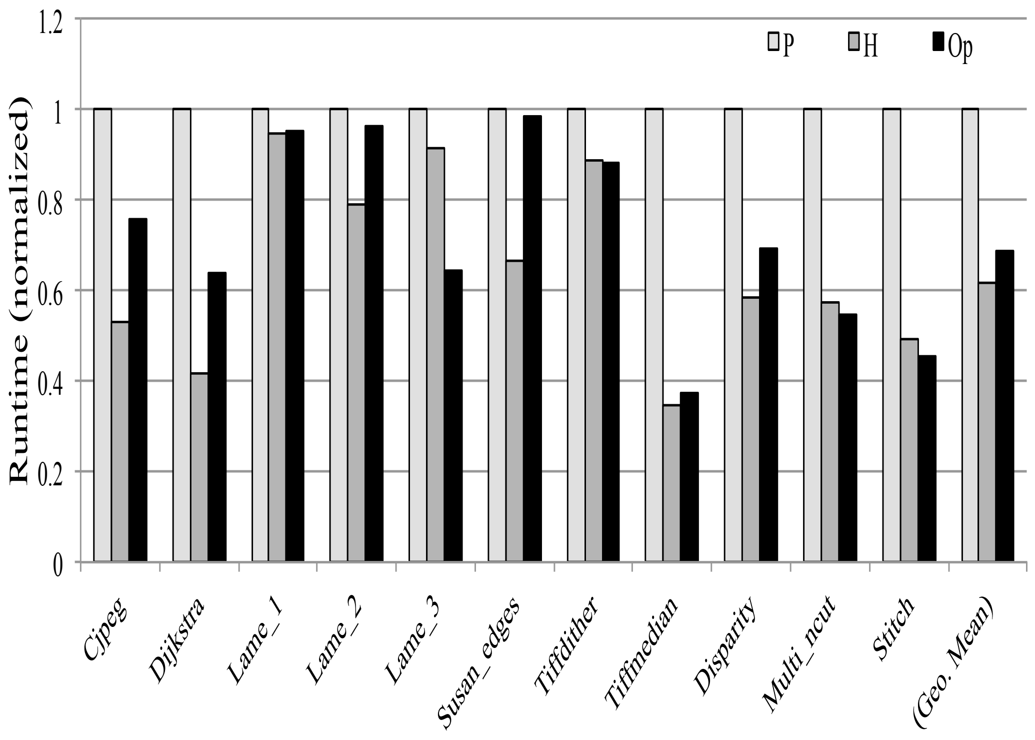


Figure 5.1: Comparing runtimes of three cases: P (predicated execution), H (our evaluator-executor transformation with hardware extension), and Op (the number of dynamic operations).

performance potential.

Fig. 5.1 compares the runtime results. First we note that our scheme with hardware extension follows the number-of-operations prediction very closely. In most cases our scheme generates performance that is similar to the number-of-operations prediction. Surprisingly, however, in several cases our scheme can excel the prediction significantly, including Cjpeg, Dijkstra, and Susan_edges. This is because the mapping result of our scheduler is not exactly proportional to the number of operations of the DFG (Data Flow Graph). Rather our scheduler, which is based on a variant of modulo scheduling [22] specialized for CGRAs, tends to perform better for smaller DFGs because smaller loops usually have lower routing pressure; to wit, scheduling two smaller loops may generate better results than scheduling one large one. The other reason is that in the original loop the iterator has to be routed from the beginning until almost the end of a loop, whereas our transformation often stores the iterator early in the evaluator loop, and therefore does not consume any routing PEs.

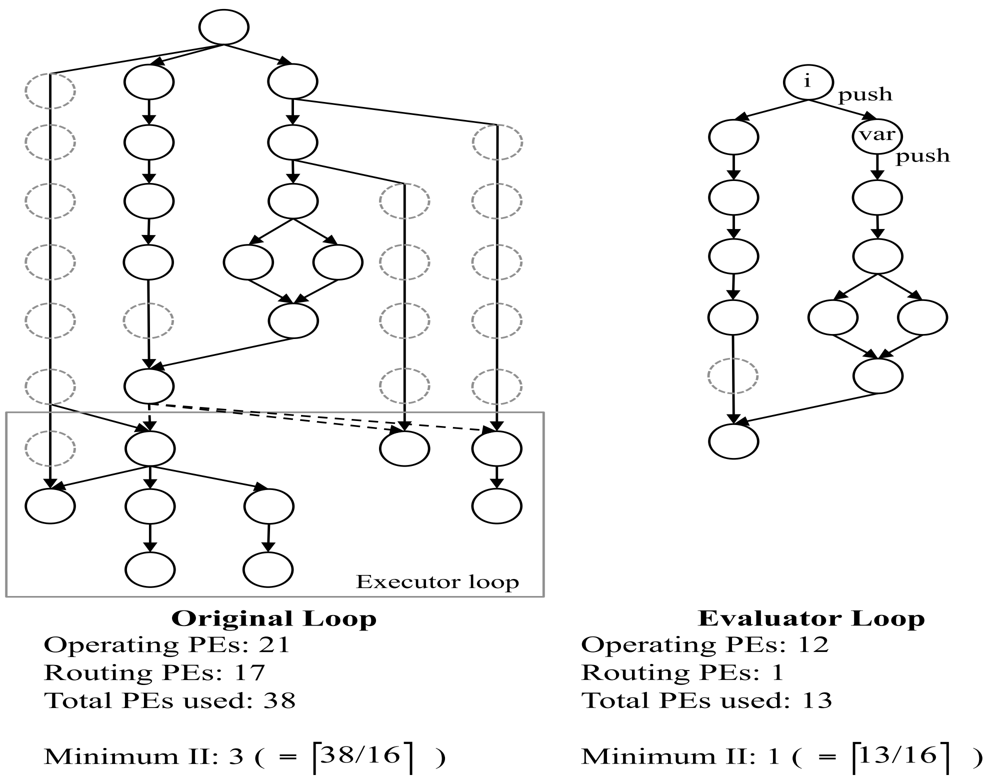


Figure 5.2: Dijkstra loop example, before and after the transformation.

To illustrate the superlinear increase of the runtime in the number of operations, Fig. 5.2 shows the DFGs of Dijkstra loop before and after the transformation. While the number of operations is reduced by 43% only (from 21 to 12), the number of necessary PEs including those used for routing only is reduced by 66% (from 38 to 13). Consequently, the minimum initiation interaval on a 4x4 PE array is reduced by 3 times (from 3 to 1).

On the other hand, in the case of Lame_3 the performance of our scheme is much worse than the prediction, and is similar to (but still better than) the original. This is a direct consequence of the evaluator, in this particular case, taking almost as many cycles as the original loop—they happen to be scheduled with the same initiation interval with the evaluator having fewer stages. Thus though reduced operation count is very likely to lead to reduced runtime, this may not always be the case, due to the randomness of the scheduling process. Overall using our hardware scheme can reduce the kernel runtimes by 38.0% on average, which outperforms the number-of-operations prediction mainly due to the simpler routing.

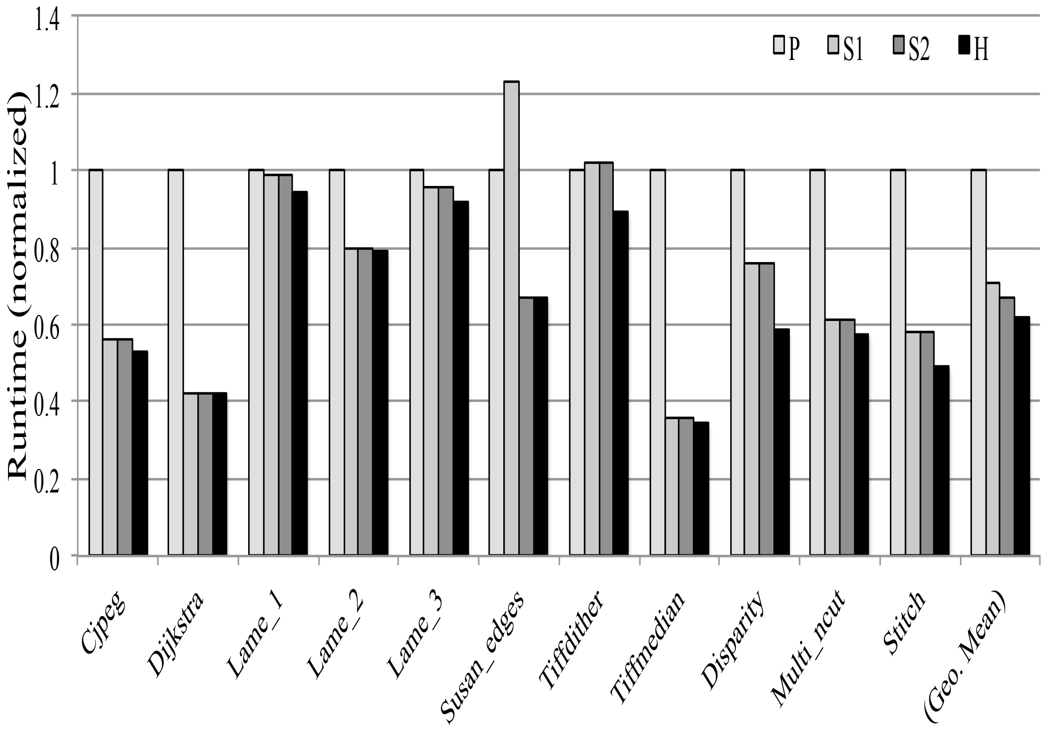


Figure 5.3: Comparing runtimes of four cases: P (predicated execution), S1 and S2 (our evaluator-executor transformation without hardware extension), and H (with hardware extension).

5.1.2 Performance of Software Transformation Only

In this section we compare our evaluator-executor transformation *with* and *without* hardware extension. Fig. 5.3 shows the runtime results, as normalized to that of the predicated execution (**P**). The rightmost bar (**H**) represents the case with hardware extension. The other two bars denote software-only cases. Their difference is that for transfer variables, **S1** exclusively uses recomputation, whereas **S2** uses the better of recomputation and save-and-restore, based on the number of operations.

From the graph we observe that while the software-only schemes can improve the runtime significantly compared with the predicated execution, the difference between software-only and hardware extension is only marginal. In particular, our software-only scheme, S2, can reduce the kernel runtimes by 33.2% on average compared to the predicated execution, which is less than 4.8% point difference from that of the hardware version. This is true even for some loops (Lame_2, Tiffmedian) with the highest number of transfer variables. That said, for some other applications the additional operations added to the software version do affect the performance rather significantly. For instance, Disparity and Stitch, going from **H** to **S2**, see 30% and 19% runtime increases, respectively.

Between the **S1** and **S2** cases, though the overall difference is small (they are only 3.8% point different in terms of the geometric mean), some applications do see huge runtime reductions. Most notably the runtime of Susan_edges is reduced by 46% from 1.23 times the baseline (when all transfer variables are recomputed) to 0.67 times (when they are all passed via memory), which is because the recomputed expressions in this loop are very large and complex. For those loops, passing transfer variables via memory can eliminate the overhead of recomputation, and generate performance improvement that is comparable to the hardware version.

5.1.3 Energy Comparison

To evaluate the energy consumption of our proposed scheme we use the energy model of [16], where the CGRA's dynamic power consists of PE (Processing Element) array power, configuration memory power, and the (CGRA's) local memory power. The PE array power has two components, individual PEs' power dissipation and that of the rest of the PE array. The most interesting parts are the individual PE's power, which is dependent on the oepration being performed (ALU, multiply, divide, routing, or idle), and the local memory power, which is proportional to the number of memory accesses. Except for the local memory power, which we obtain from Cacti 6.5 [25] for a 65 nm technology, we use the power numbers of [16], scaled to the 65 nm technology. The leakage power is assumed to be 50% of the dynamic power with PEs being idle whereas the leakage power of the local memory is obtained from Cacti. We assume that the CGRA runs at 500 MHz.

We compare the total energy consumption of the CGRA and its associated memories in two cases: using predicated execution only (**P**) vs. using our software-only scheme (**S2**). Though faster for many loops, our software-only scheme always generates more memory accesses than the predicated execution. This is because in the predicated execution load/store operations that are disabled

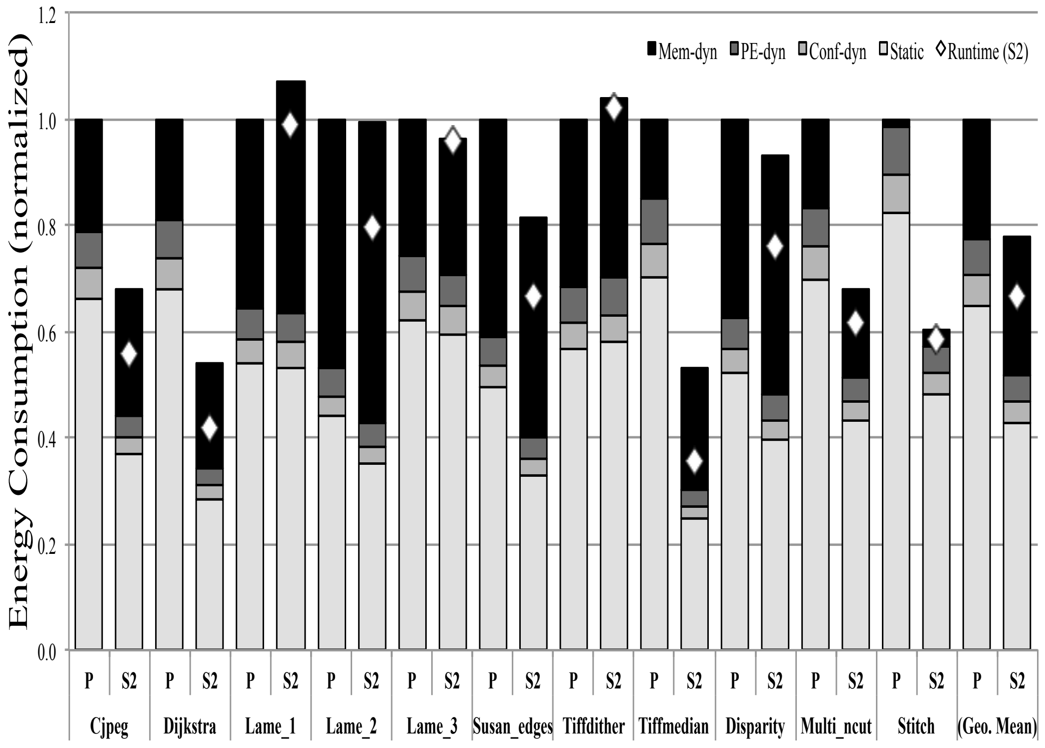


Figure 5.4: Comparing energy of predicated execution (P) and our software transformed loops (S2).

due to predication do not generate memory operations whereas our scheme generates loops that have extra load/store operations due to the iterator and possibly one or more transfer variables. In both the cases the underlying hardware is the same.

Fig. 5.4 compares the energy results, overlaid with the normalized runtime results of the **S2** case. The energy consumption is broken down into different categories, among which the leakage accounts for the largest portion on average. In our energy model, only the local memory energy and the PE array energy are affected by the operations performed on the CGRA. The others are proportional to the runtime, and in those categories we see reductions of about 33% on average going from the baseline to our software scheme. The PE array's dynamic energy is only slightly larger than that of the configuration memory, and is overall reduced by our scheme, which is largely a by-product of the reduced runtime. Lastly the local memory's dynamic energy, which accounts for 22.4% in the baseline, is increased by 18.2% (to 26.4% of the baseline total energy) due to the additional memory operations generated by our software transformation. However, this increase is not excessive in any application. This is because the iterator and optionally transfer variables that are saved in the evaluator loop are *only for those iterations that will be executed in the executor loop*. In other words, we save only the data that will be used later, which helps keep down the energy price we pay in our transformed loops. Overall our software technique can reduce the energy consumption of the entire CGRA including configuration and data memory by 22.0% on average.

CHAPTER VI

Conclusions

Many conditionals found in loops have unbalanced branches in terms of both size and execution frequency. Interestingly, often larger branches are less frequently executed, presenting an opportunity for optimization unknown to predicated execution, which blindly allocates resources regardless of the execution frequency. For such loops with unbalanced conditionals, we present a software technique that divides a loop into two or three smaller loops so that the condition part is evaluated only in the first loop while the less frequent but larger branch is executed in the second loop in an efficient way. To reduce the overhead of extra data transfer caused by the loop fission, we also present a hardware extension for a class of coarse-grained reconfigurable architectures (CGRAs). Our experiments using MiBench and computer vision benchmarks on a CGRA demonstrate that our techniques can improve the performance of loops over predicated execution by up to 65%, or 38.0% on average when the hardware extension is used. Without any hardware modification, our software-only version can improve performance by up to 64%, or 33.2% on average, while also reducing the energy consumption of the entire CGRA by 22.0% on average.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 5

[2] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. *SIGARCH Comput. Archit. News*, **40**[3]:49–60, June 2012. 5

[3] R. Camposano. Path-based scheduling for synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **10**[1]:85–93, 1991. 5

[4] João M. P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer, 2009. 4

[5] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on CGRAs. In *FPT*, pages 285–292, 2012. 1

[6] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers.

In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 477–488, New York, NY, USA, 2011. ACM. 5

[7] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient simt control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society. 5

[8] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 12–23, New York, NY, USA, 2011. ACM. 5

[9] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 269–272, New York, NY, USA, 2001. ACM. 5

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. 2, 19

[11] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on CGRAs. In *Proceedings of the 49th Design Automation Conference (DAC)*, 2012. 1

[12] Kyuseung Han, Jong Kyung Paek, and Kiyoung Choi. Acceleration of control flow on CGRA using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429–432, 2010. 1, 5

[13] Kyuseung Han, Seongsik Park, and Kiyoung Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1367–1372, 2012. 5

[14] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 642–649. IEEE Press, 2001. 4

[15] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, pages 550–556, 1995. 5

[16] Yongjoo Kim, Jongeun Lee, Toan X. Mai, and Yunheung Paek. Improving performance of nested loops on reconfigurable array processors. *ACM Trans. Archit. Code Optim.*, **8**[4]:32:1–32:23, January 2012. 1, 24

[17] Jongeun Lee, Kiyoung Choi, and N.D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *Design Test of Computers, IEEE*, **20**[1]:26–33, 2003. 4

[18] S.A. Mahlke, R.E. Hank, J.E. McCormick, D.I. August, and W.-M.W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 138–149, 1995. 1

[19] Bingfeng Mei, Serge Vernalde, and Diederik Verkest. ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, number 2778 in Lecture Notes in Computer Science, pages 61–70. Springer Berlin Heidelberg, January 2003. 15, 19

[20] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, Washington, DC, USA, 2003. IEEE Computer Society. 4

[21] Alexander Paar, ManuelL. Anido, and Nader Bagherzadeh. A novel predication scheme for a simd system-on-chip. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, **2400** of *Lecture Notes in Computer Science*, pages 834–843. Springer Berlin Heidelberg, 2002. 1

[22] Hyunchul Park, Kevin Fan, and Scott Mahlke. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *In Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, 2008. 1, 4, 19, 21

[23] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, 1994. 4, 14

[24] Minsoo Rhu and Mattan Erez. Capri: prediction of compaction-adequacy for handling control-divergence in gpgpu architectures. *SIGARCH Comput. Archit. News*, **40**[3]:61–71, June 2012. 5

[25] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. *HP Laboratories, April*, **2**, 2008. 24

[26] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society. 2, 19